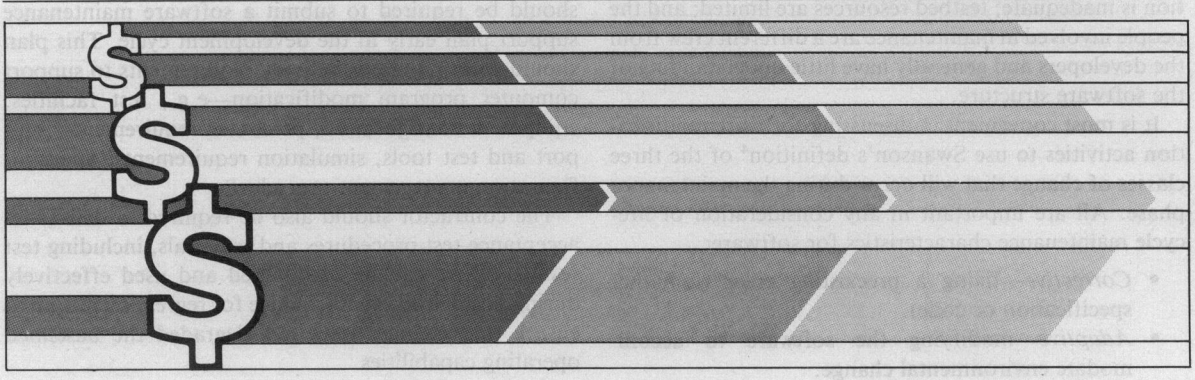
Software modifiability doesn't just happen. It must be designed, tested, and paid for just like any other software requirement.



Special Feature:

Software Maintainability: A Practical Concern for Life-Cycle Costs

John B. Munson
System Development Corporation

The issue of software maintenance cost is rapidly becoming critical to EDP users. The shortage of programming personnel, coupled with the staggering load of existing software applications being carried forward (the value of existing software is commonly estimated at $200 billion or more), means that more and more effort is going into keeping existing software working, as opposed to developing new applications.

Thirty years into the computer revolution, people still think of software as a black box which, if built with quality and tested exhaustively, can be used forever without modification. The reality is quite different, of course. First, error-free software is currently impractical (that is, uneconomical) to build. Second, almost all software is modified continuously during its useful life cycle: to add features, to expand capacities, to implement new capabilities, or to support different equipment.

Recent estimates[1] by the Department of Defense state that 60-70 percent of DoD's software dollars are being spent *after* the software has been tested and delivered by the development contractor. With an annual DoD expenditure of $3 billion for software,[2] this area of postdelivery maintenance becomes a prime candidate for major technological effort with potentially high return on investment.

However, even without resorting to technological innovation or R&D investment, there is much that can be done to reduce life-cycle costs, provided that the acquisition agencies understand the issues and are willing to spend the required time and effort (money) *during the development phase.*

What is software maintainability/modifiability?

First, we have to understand the problem. "Maintenance" is really a misnomer relative to software. In hardware vernacular, maintenance means repair, and repair means to return a piece of hardware to its state prior to a failure (e.g., restoration). On the other hand, a software failure is the exposure of a previously existing incorrect state; therefore, a repair changes the prior state. This is change, not restoration. As a matter of fact, because software is insubstantial (i.e., not physical), it does not degrade through use; it neither deteriorates over time, nor does it rot, mildew, or burn out. Once software is correct, it will be correct forever (unless modified). Therefore, as latent defects are discovered and fixed (hopefully without degrading something else), software reliability improves. One is tempted to generalize that software (like wine but unlike hardware) improves with age.

The subject of failure repair, however, is just a small part of postdelivery "maintenance" activities. Studies show that coding errors only account for 30 percent of the postdelivery discrepancies.[3] The balance are occasioned by a mistake in design or specification and, although the code performs exactly as designed, this wasn't what was originally desired operationally.

Finally, even if the software performs exactly as desired, the world changes—and that means new functions in the system for the software to perform and new equipment with which to interface. One of the apparent strengths of using a computer is the flexibility of software

0018-9162/81/1100-0103$00.75

to handle change without rebuilding or replacing a complete system. In practice, this flexibility is limited and very expensive. Why? Because modifying software is a complex, error-prone process; maintenance documentation is inadequate; testbed resources are limited; and the people involved in maintenance are a different crew from the developers and generally have little understanding of the software structure.

It is most convenient in discussing software modification activities to use Swanson's definition[4] of the three classes of change that will occur during the maintenance phase. All are important in any consideration of life-cycle maintenance characteristics for software:

- *Corrective*—fixing a preexisting error (in either specification or code).
- *Adaptive*—modifying the software to accommodate environmental change.
- *Perfective*—improving or augmenting the performing capabilities.

And whereas software doesn't degrade over time, there is reason to believe that the act of maintaining software degrades its future maintainability. IBM studies on OS,[5] for example, show that over its lifetime OS becomes uneconomical and, consequently, impractical to maintain.

Modifiability must be built into software from the start.

Clearly, modifiability must be built into software from the start. Though "good programming practices" and "modern programming techniques" are necessary, they are not sufficient. Rather, modifiability must be explicitly planned, recognizing that it costs money in the development process and cannot be left to chance. And it must be managed from start to finish.

Development phase techniques

Modifiability starts with the contract. Although much good R&D work[6] is being done to define and quantify such software attributes as quality, reliability, and modifiability, this effort has not progressed to the point where such capabilities can be directly and meaningfully contracted for. Contracts must still be written to require software performance characteristics that can be empirically measured and evaluated. The contract can, however, define modifiability as a design goal and can require that the contractor submit a contractually binding development plan which addresses how the techniques discussed below will be built into the software. It can also require that the contractor produce approved programming standards to which his staff must adhere in designing and writing the code.

Equally important are the factors that must be excluded: Use of proprietary software that will not be delivered should be avoided. Use of off-the-shelf software not built or documented for maintainability must be carefully examined and avoided unless it is cost-effective in a total life-cycle sense. Likewise, the use of assembly language instead of higher-order language must be carefully evaluated (and in most cases discouraged).

In addition to the development plan, the contractor should be required to submit a software maintenance support plan early in the development cycle. This plan should specify the postdelivery requirements to support computer program modification—e.g., test facilities, equipment configuration, personnel requirements, support and test tools, simulation requirements, and configuration management procedures.

The contractor should also be required to deliver all acceptance test procedures and materials, including test results. These can be maintained and used effectively during postdelivery maintenance for regression testing to ensure that changes have not degraded the baselined operating capabilities.

Finally, the contract must require the proper software documentation for use in postdelivery maintenance.

If modifiability standards are specified, the initial development will cost more: first, in an increased contract price, and second, in the inefficient use of computer resources in the operational system (core space and operating speed). However, in the long term this is a wise investment.

Remember, the contract is the binding document. The contractor is only required to deliver what was explicitly contracted for; therefore, if the customer wants modifiability the contract must explicitly require it. To leave modifiability to the contractor's promises and good intentions is to pay lip service to the idea with little or no chance that anything constructive will result.

The requirements specification. Whether the requirements specification is developed before the contract is let or is included and prepared under the development contract, it is still the place to define the modifiability baseline. The key modifiability issue in the requirements document is system expandability. The customer and contractor both should look to the future, asking the following questions:

- Which of today's limits and capacites may change?
- How are equipment or resources likely to evolve?
- Which parameters should be flexible and which firm?
- How much spare capacity in the equipment for growth should be built into the specification?

This whole area can, of course, be taken to an extreme. Therefore, careful management control, cost/benefit analyses, and tradeoff studies must be used in the process to ensure practicality.

Designing flexible user interfaces. Generally, all complex data processing systems have an extensive and sophisticated set of on-line displays and reports associated with their processing requirements. Experience has shown that this area of interface with the users of the system is where rapid and continuous change can occur. For printed output, general-purpose report generators are a fairly common commercial item and can provide almost unlimited flexibility. In the event this option is

impractical, reports should at least be programmed so that changes in content or form do not require a complete reprogramming. "Table-driven" programming techniques can be used effectively in this area.

For CRT displays, which interact directly with the user, more sophistication can be obtained. The goal should be to allow users to construct or modify displays from the console without having the programmers modify computer programs. Again, if this option is too expensive or sophisticated, the software must be designed and written to give parameters for display formats and structures, minimizing coding and checkout activities.

Building modifiable computer programs. The keys to easily modifiable software are clarity and simplicity in both the design and the implementation. Clarity and simplicity are enhanced by modularity and module independence, by structured code, and by top-down design and implementation, among other techniques. It is important that an adequate methodology be defined and specified early in the design phase and that its use be enforced throughout the implementation.

The keys to easily modifiable software are clarity and simplicity in both design and implementation.

The allocation of functional requirements to elements of code (discrete computer programs) represents an important step in the design process that critically affects modifiability. The following general guidelines for the definition of modules will have an extremely positive impact on modifiability and should be considered for inclusion in the established programming standards.[7-10]

Modifiability: how to get it

Software modifiability doesn't just happen naturally. It must be designed, tested, and paid for just like any other software requirement. The contractor has to commit to a plan for achieving that objective. Here are the steps for reviewing the implementation of the plan. Remember: without a contractually binding plan to control against, little will be accomplished.

Program standards documentation. Early in the program development cycle, the contractor should be required to submit for customer review the project-specific programming standards he intends to employ. The customer should verify at this time that when the standards are implemented, they will support his maintainability goal. Further, the mechanism by which the contractor will enforce and manage these standards should be specified. Most standards can be verified by using a computer program which analyzes source code for adherence. One note of caution, however: such an analyzer can tell you the program has comments, but it can't tell if the comments are meaningful. Only technical inspection can do that.

System-level design reviews. The purpose of a system-level design review is to evaluate and monitor the progress and technical adequacy of the selected design approach for the software. The review should emphasize design, language usage, and programming standards. Although this is a design review, it should also be used to review the developer's planned implementation methods, as described in the development plan. The reviewer should check for the following features, which facilitate the development of modifiable software:

- Will the software be designed in a manner that provides for ease of modification, as planned for in the development plan?
- Have variations from the top-down or proposed hierarchical structure been justified? (Real-time requirements, for example, may dictate other than a top-down approach.)
- Have computer program modules and data-base interfaces been defined so that independent, detailed design can be started at a lower level? Have interfaces been defined in a simple and explicit manner?
- Have functions and subfunctions been allocated to modules in a way that enhances modularity and functional independence?
- Has the software data base been defined in a symbolic manner?
- Is there a centralized, machine-processible, data-definition capability? If not, is there a manual procedure established to define and control the data-base definitions?
- Have all areas where assembly language is to be used been identified and justified?
- Are programming standards with coding examples available to the programming staff? Do the standards cover techniques for designing and developing modular and structured software? Have methods for improving module independence been included? Have procedures been established for enforcing the standards (e.g., program walk-throughs, code audits, and automatic verifiers)?
- Have programming personnel been trained in the concepts of structured programming, operating system requirements, library procedures, and modular coding techniques? If not, is a training program scheduled for all current and newly assigned personnel?
- Is an industry-standard, higher-order language being used? If not, is the developer's HOL selection based on reasonable technical consideration?
- Will the compiler be tested and validated before it is required for coding?
- Is structured code to be used? If the compiler does not support structured programming, is a preprocessor available?
- Have all performance requirements been allocated to modules? Is there a traceability matrix which relates requirements to their implementation in the modules?
- Have support tools been defined? Have tools that require new development been designed? Have they been designed in a modular manner? Have debugging tools been defined? Have all modifications to

- Use hierarchical module control structures whenever possible.
- Achieve singularity of function by assigning single or closely related functions to a module and avoiding conglomeration.
- Limit module size: some say to the equivalent of one page of listing (i.e., 50-60 statements), others say up to 200 statements, but *all* say limit!
- Make sure each module has only one entrance and exit. (An exception may be made for error conditions.)
- Write each module so it does its own housekeeping as its first act.
- Isolate machine interfaces by assigning machine interface functions to a few controlled routines. Use global data structures to define peripheral characteristics. Use "virtual" instead of physical peripherals.
- Control or partition data-base access by the modules. Limiting the number of functions in a module should help this, but care must be exercised to limit how many files each module uses and sets.
- Reduce interface/communication complexity by passing parameters directly between modules. Set standard calling sequences.
- "Hide" internal processing characteristics of the module from external view. Don't base intermodule communication on "how" a module works, but on what it is to do. Don't use implicit knowledge of one module in another.
- Use a global data-base adaptation table for system configuration, data limits, general parameters, etc. Do not hard code system-level parameters into the modules.
- Use "go-to-less" or structured programming logic.
- Avoid self-modifying code, embedded constants and parameters, complex expression of math statements, and absolute or relative addressing.

commercial off-the-shelf or existing debugging aids been identified? (Eventual customer ownership of support tools and related documentation should have been established in the contract, but the plan should be reviewed at this time.)
- Are the test requirements for modifiable software included in the updated test plan? In general, software design features, which provide for ease of modifiability, must be reviewed by inspection.

Detailed design review. A detailed design review provides a formal technical review, or series of reviews, at completion of the detailed design of each group of related computer program modules. Successful completion signifies verification of the detailed design and allows initiation of code and test activities.

The reviewed design detail should be explicit enough for the individual program modules, their interfaces, and associated data-base requirements to be clearly identified and then coded.

The emphasis during the review should be on the program structure, modularity, language usage, programming standards, support tools, data-base design, interfaces, and planned coding techniques. A traceability matrix should be available that further relates the requirements directly to the implementing modules. In reviewing the developer's design, the reviewers should check for the following additional features, which facilitate the development of maintainable software:

- Have all software modules been specified?
- Are all module interfaces defined and documented in accordance with the design philosophy stated in the development plan? Are all control data to be passed via the interfaces clearly defined, and has an attempt been made to minimize this type of data?
- How well have the modules retained their independence?
- Has machine dependency been minimized (e.g., not overly dependent on word size, peripherals, or storage characteristics)?
- Has the system data base been designed and documented? Has it been symbolically defined and referenced (e.g., was a central data definition used)?
- Is I/O centralized and separate from computation functions?
- Are module source code estimates within the developer's size limitations?
- Are modules functionally cohesive (i.e., limited to a single or small number of closely related functions)?
- Have any modules requiring assembly language been identified and justified?
- Have standards been established to limit the sequences of code to a limited set of control structures?
- Have all support tools specified for coding and debugging (e.g., pre- and postprocessor) been produced? If not, are they scheduled early enough to meet the needs of the development schedule?
- Have all modules been designed to have single entry and exit points (with the exception of certain computer interrupts and erroneous condition exits)?
- Whenever initialization or housekeeping is required, will those functions be internal to the module requiring them?
- From a recovery point of view, computational or I/O modules should not abort. They should pass an error condition back to the main control level which is designed to make abort or recovery decisions. Has this concept been followed?
- Does the program support library reflect the software structure? Have procedures and libraries been established to control the source and object files and listings as they are produced?
- Do the test procedures include inspection for maintainable software requirements (e.g., module size, structured code, HOL and assembly language, and module independence)?
- Will a code auditor be available at compile time to monitor software characteristics and enforce standards?

Delivery and acceptance. Prior to delivery and acceptance, the customer and developer should jointly review the materials for conformance with the specified maintainability goals:

Control of data. In large systems, control of the data-base structures, both global and local, is essential to good program development and, later, to enhancing modifiability. The best way to control and handle data definition and use during design and programming is to have all data centrally controlled and symbolically defined and referenced, and to have this definition (dictionary) machine-processible. This capability exists today to some extent in the compilers for most higher-order languages. However, if a master-independent dictionary file can be created (like the Jovial-compool capability), then the benefits are significant in terms of system control and simply implemented tools. This physical representation of data structures is the single most powerful tool for improving both implementation and maintainability characteristics. For example, data bases can be reshuffled, extended, or reduced, and these changes need only be made in the central dictionary. Then the programs are recompiled to have a reorganized system. Also, data recording and reduction programs can be operated symbolically, presenting outputs in natural language properly tagged and scaled. Furthermore, "set-used" listings can be generated showing precisely how each module interfaces with the data base.

Not all contracts will embody all of these principles. However, even limited methods of data definition, such as those Fortran provides, can be developed into strong tools with a little creativity. What is critical is that *all* data be defined symbolically, documented, and controlled.

Postdelivery techniques

Equally as important as the early planning for how the development effort will be accomplished is early planning for how maintenance will be facilitated. Tools, techniques, documentation, and controls that will be used during the operations and maintenance phase must be considered and planned even before the development effort begins.

Early planning for maintenance is as important as development planning.

Documenting for software maintenance. One of the current problems with postdelivery maintenance of software is the quality of its documentation. Most documentation ranges from nonexistent to out-of-date. DoD standards require extensive program-level descriptive documentation, called Part II specifications (C5 specs), which contain detailed program descriptions and flow charts. Commercial practices generally require much the same information as the military, but usually are not as extensive. Unfortunately, even if this documentation is good when delivered, it will invariably and quickly fall out of date as the product is modified. Further, it is of limited usefulness even if it could be maintained perfectly. This is an unfortunate situation since at least 50 percent of the total documentation costs (and they are extensive) go to producing this set of documents.

It is clear that the scope and form of the documentation to support modification must be rethought in light of the problem we're trying to solve. What are the facts of software maintenance activities, and what are the real needs?

People. What do we know about the *people making the changes?*

(1) They are usually less experienced professionally than the implementors.
(2) They have some competence in the language in which the system is written.
(3) They have a general knowledge of the total system but only superficial knowledge of individual computer programs.
(4) Each has responsibility for prodigious amounts of code.

Their task. Problems (either error identification or modification requests) are stated to them in operational

- Review the test results to ensure adequate inspection of the code to determine the proper use of the specified maintainable software attributes. Check module size, language, structured code, adherence to programming standards, and code readability by reviewing library records and/or doing spot checks.
- Review the code to ensure that the following difficult-to-maintain features have been avoided:
  - — self-modifying code,
  - — absolute addressing,
  - — embedded constants and literals, and
  - — relative addressing.
- Have design changes, requirement changes, and error corrections been made without major impacts on the software structure?
- Have the traceability matrix and the test procedures been updated to reflect design and requirement changes?
- Have portions of the software that are time- or space-sensitive been identified and documented? Have adequate comments been made on these portions of the code to alert maintenance programmers?
- Is system-level task documentation current and available?
- Are the program source code listings readable and reasonably self-documented?
  - — Are they adequately commented?
  - — Can they be easily reviewed?
  - — Is it clear what each area of code is intended to do?
  - — Are the data references symbolic and are the acronyms meaningful?
  - — Are the dates and versions of the listings the same as those in the developer's list of validated materials?
- Have all development and test support tools been found acceptable? Make sure the developer validates all tools to be delivered.
- Are all test procedures, materials, and results available?
- Is a current inventory of all computer program materials available?

terms, which they convert to software system implications and then to specific program modifications.

How does this translate to documentation needs?

First, let's eliminate Part II specifications and their equivalents. Since all the code exists and since our maintenance programmers must understand the programming language anyway, the code represented by listings is absolutely the best, most maintainable form of documentation. Only a fraction of the cost of formal documentation would be required to upgrade listings to the state of first-class documentation. The only caution is that discipline must be specified and enforced during program development to ensure adequate comments in the listings. The boxed material provides a sample outline for a program listing annotation scheme designed to fit maintainability needs.

Adequate listings, however, are only half the problem. What we need to supplement listings are overview software system descriptions organized in terms of operational capabilities. This is the documentation of the task processing strings used by many professional software houses in their design process. The document should specify the flow for each operational task, and the description should include only the in-context references to the programs and data-base elements involved in processing each task. During maintenance, then, when a problem is noted, the analyst checks the task listing which references the routines and data elements involved and then goes to the listings of the specific programs involved for detailed information.

Testbeds and support computer programs. The scope and extent of potential maintenance activities must be thoroughly evaluated prior to awarding a development contract. They must also be reevaluated continuously to be sure that adequate system resources are going to be available to make maintenance activities cost-effective. The first point here—one commonly overlooked—is the availability of equipment for test and installation of changes. What normally happens is that operational resources are assumed to have spare capacity without any realistic evaluation. Quite naturally, maintenance never achieves the necessary priority while a user is trying to get a new operation settled in, since operational use often exceeds expectation. The result is that the installation of changes is slowed and the very expensive personnel maintenance resources are wasted through inefficient use of their time. With the currently plummeting cost of hardware, consideration should be given to replication of the facility for both backup and maintenance purposes. Failing this, enough spare capacity should be procured to allow maintenance time even in a "worst load" situation.

Review the contractor's programming standards early in the development cycle.

Next, the more realistically the test environment can simulate the operational environment, the more confidence can be achieved that validation of changes will avert potential disruption of operational capability following changeover to the modified system. Either equipment or software simulation capability should be procured to provide a realistic test environment. As with other items mentioned, a competent system engineering cost/benefit analysis must be performed to ensure proper tradeoffs between cost and risk. Until the customer has such a study, he is dealing with a very intangible area in which history shows that normal intuition will severely underestimate the problem.

Finally, during software procurement, the contract should require, as a minimum, delivery of all test tools used during development (with documentation and user manuals). Further, if a high level of maintenance activity is anticipated, the investment in supplemental tools will be well worthwhile. Such items as

- data generators and "compool" assemblers,
- environment simulation programs,
- program support library,
- code auditors,

Criteria for documenting source-code listings

Program code should contain sufficient information to determine or verify the code's objectives, assumptions, constraints, inputs, outputs, components, and revision status. Comments should meet the following checklist:

- Each computer program module contains a header block of comments which describe the following:
  - — computer program name,
  - — effective date (last revision),
  - — accuracy requirements,
  - — purpose,
  - — limitations and restrictions,
  - — modification history (a list of changes added by date),
  - — inputs and outputs,
  - — methods used or algorithm definitions,
  - — assumptions, and
  - — error recovery types and procedures for all error exits.
- Any intermodule communication should be clearly specified by comments.
- Decision points and subsequent branching alternatives should be adequately commented upon and properly indented to show the block structure of the coding logic.
- Comments should adequately explain the implementation logic for each program algorithm.
- Variable names should describe the physical or functional property represented or explained in the comments.
- The comments should define all parameter ranges and their default conditions.
- The function of the module and its use of inputs/outputs should be sufficiently described to facilitate module testing.
- Comments should be provided to support selection of specific input values to permit performance of specialized program testing.
- When known, information should be provided to support assessment of the impact of a change in other portions of the computer program.
- All computer program statements which have undergone modifications (after baselining) should have an identification number included that associates the change with the documented change request.

- memory dumps and interpretive trap and trace,
- break point or memory change analysis,
- on-line diagnostics,
- data recording and reduction, and
- test data verifiers

should be evaluated against potential future use. They may be valuable in both the development and maintenance phases. If it is decided to build unique test tools, their delivery should be specified in the contract. Excellent surveys of test tools[11,12] are available for use in selecting aids to support postdelivery maintenance activities.

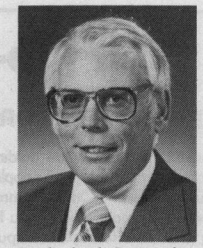
Controls and standards. It is important that the postdelivery modification activity adhere to all the basic design and implementation principles used by the development contractor. If not, then, like OS, the system will gradually lose its inherent easy modifiability. Likewise, it is equally important that the same strong configuration management disciplines enforced early in the program be extended into the maintenance phase. To aid this process, the developer should be required to deliver his internal programming standards and his configuration management plan, and the maintainer should be required to adapt them to the maintenance environment and to continue their implementation and enforcement.

Even though this apparently straightforward and pedestrian approach to modifiability is well within the current state of the art, it is clearly being ignored in most of today's applications.

In this age of automation, we seem to be waiting for maintainability to be automatically produced from our automated systems. It will in fact come about only as a result of the proper recognition of the cost benefits and the successful implementation of these first elementary efforts to achieve it. ■

References

1. B. C. DeRoze, "The United States Defense Systems Software Management Program," *Proc. AIAA Government Initiatives in Software Management,* Conference-II, 1976.
2. *Ibid.*
3. B. W. Boehm et al., "Some Experience With Automated Aids to the Design of Large-Scale Reliable Software," *IEEE Trans. Software Engineering,* Vol. SE-1, No. 1, March 1975, pp. 125-133.
4. E. B. Swanson, "The Dimensions of Maintenance," *Proc. 2nd International Conference on Software Engineering,* October 1976, pp. 492-497.
5. L. A. Belady and M. M. Lehman, "A Model of Large Program Development," *IBM Systems Journal,* Vol. 15, No. 3, 1976, pp. 225-252.
6. B. W. Boehm et al., "Quantitative Evaluation of Software Quality," *Proc. 2nd International Conference on Software Engineering,* October 1976, pp. 592-605.
7. Douglas T. Ross et al., "Software Engineering: Process, Principles and Goals," *Computer,* Vol. 8, No. 5, May 1975, pp. 17-27.
8. G. M. Weinburg, *The Psychology of Computer Programming,* Van Nostrand Reinhold Company, New York, 1971.
9. E. Yourdon, *Techniques of Program Structure and Design,* Prentice-Hall, New York, 1975.
10. *Software Acquisition Management Guidebook: Software Maintenance,* System Development Corporation, Air Force Electronic Systems Division, ESD #TR-77-327.
11. D. J. Reifer, "Automation Aids for Reliable Software," SAMSO Report TR 75-183, Aerospace Corporation, El Segundo, Calif., August 1975.
12. E. Miller, *Tutorial—Automated Tools for Software Engineering,* IEEE Computer Society, Los Alamitos, Calif., November 1979.



Jack Munson is vice president for software engineering at System Development Corporation in Santa Monica, California. He is responsible for the company's software development activities in addition to directing its extensive software engineering R&D program. He has a BS degree in mathematics, is a senior member of IEEE, and is on the Executive Board of the IEEE Computer Society's Software Engineering Technical Committee.