tham, MA, 1973.
[10] S. Hori, "CAM-I long range planning final report for 1972," ITTT Res. Inst., Dec. 1972.
[11] L. J. Peters, *Handbook of Software Design*. New York: Yourdon Press, 1980.
[12] S. L. Pollack, H. T. Hick, Jr., and W. F. Harrison, *Decision Tables, Theory and Practice*. New York: Wiley-Interscience, 1971.
[13] M. Montalbano, *Decision Tables*. Palo Alto, CA: Science Research Associates, 1974.
[14] M. Hamilton and S. Zeldin, "Top-down, bottom-up, structured programming and program structuring," Charles Stark Draper Lab., Massachusetts Institute of Technology, Cambridge, MA, Document E-2728, Dec. 1972.
[15] J. Rood, T. To, and D. Harel, "A universal flowcharter," in *Proc. NASA/AIAA Workshop on Tools for Embedded Computer Systems Software* (Hampton, VA), pp. 41-44, Nov. 1973.
[16] I. Nassi and B. Shneiderman, "Flowchart techniques for structured programming," *SIGPLAN Notices*, vol. 8, no. 8, pp. 12-26, Aug. 1973.
[17] N. Chapin, R. House, N. McDaniel, and T. Wachtel, "Structured programming simplified," *Comput. Decisions*, pp. 28-31, June 1974.
[18] N. Chapin, "New format for flowcharts," *Software Practice Experience*, vol. 4, pp. 341-357, 1974.
[19] P. G. Hebalkar and S. N. Zilles, "TELL; A system for graphically representing software design," IBM Res. Rep. RJ 2351, Sept. 1978.
[20] T. Demarco, *Structured Analysis and System*. New York: Yourdon, 1978.
[21] J. D. Warnier, *Logical Construction of Programs*. New York: Van Nostrand-Rheinhold, 1977.
[22] M. A. Jackson, *Principles of Program Design*. London, England: Academic Press, 1975.
[23] R. C. Linger, H. D. Mills, and B. F. Witt, *Structured Programming Theory and Practice*. Reading, MA: Addison-Wesley, 1979.
[24] H. F. Ledgard, "The case for structured programming," *Nordisk Tidsrift for Informations Behandling, Sweden*, vol. 13, pp. 45-47, 1973.

Software Quality Assurance: Testing and Validation

JOHN B. GOODENOUGH AND CLEMENT L. McGOWAN

Abstract—**There are many pitfalls for the unwary hardware engineer who must develop software. In particular, hardware quality control procedures and concepts can easily be misapplied. The purpose of this paper is to help hardware-oriented engineers apply some of the quality assurance lessons learned by software engineers. We first discuss how software is and is not analogous to hardware, and then outline recommended software engineering approaches for developing high-quality software products. We concentrate on methods for preventing and detecting software errors.**

I. INTRODUCTION

A. The Software-Hardware Analogs

THERE ARE many similarities between software and hardware design and development, but unless one is careful, the similarities can be misleading. For example, hardware failures are sometimes considered analogous to software failures, but often the wrong analogies are drawn. In particular, component deterioration is the usual cause of hardware failure. In contrast, software failures are almost always *design errors* that show up only when the software is used under certain conditions.

For example, a hand calculator was once manufactured that did not correctly compute the sine function for all argument values. All units computed the same incorrect value for particular arguments. This was clearly a design error—the circuit for calculating sines was incorrectly designed. Since the calculator gave correct answers for most arguments, the problem was only discovered by a few users. This is typical of design errors, which are latent until the hardware or software is exercised under the appropriate conditions.

Because software errors are analogous to hardware design errors, software quality assurance focuses on techniques for getting the design right. Adapting hardware quality assurance procedures to software development means focusing on the procedures used to prevent and detect design errors. This is the proper analogy between software and hardware quality assurance. It illustrates our point that unless one makes the proper analogies, intuitions about how to ensure hardware quality will lead one astray when working with software.

A comparison of hardware and software life cycles is given in Fig. 1. This figure indicates the phases of developing a new hardware or software product. Although Fig. 1 provides an

Manuscript received February 12, 1980; revised April 14, 1980.
The authors are with SofTech, 460 Totten Pond Road, Waltham, MA 02154.

0018-9219/80/0900-1093$00.75

	HARDWARE	SOFTWARE
DEVELOPMENT	Determine User Requirements	Determine User Requirements
	Develop Product Concept (Functional Design)	Develop Product Concept (Functional Design)
	Specify Component Design (Detailed Design)	Specify Component Design (Detailed Design)
	Build and Test Prototype	*Implement and Test Programs
	Develop Manufacutirng Techniques	--
INSTALLATION	Manufacture Product	*Copy Programs
	Make Product Available to User	Make Program Available to User
MAINTENANCE/ IMPROVEMENT	Maintenance (Correct Component Failures)	--
	Recall Product to Correct Design Flaws	*Maintenance (Correct Implementation and Design Errors)
	Enhance Product	Maintenance (Provide Enhanced Capabilities; Adapted to Changed User Environment)
PHASE-OUT	Unit is Unusable and Unrepairable (replace with new unit)	--
	Product is Obsolete	Product is Obsolete

*marks important differences in terminology

Fig. 1. Corresponding steps in software and hardware product life cycles.

oversimplified and idealized description, it shows clearly that similar terms in the two fields sometimes have radically different meanings. Perhaps the most important differences are

1) coding programs is not equivalent to manufacturing a product;
2) maintenance refers to quite different processes (see Lehman [16] for further discussion of this point);
3) program development and test is conceptually similar to developing and testing a hardware prototype, but in software, the "prototype" is the first system that gets delivered to users.

The focus of this paper is software testing and quality assurance. As Fig. 1 indicates, we will therefore be discussing procedures and concepts aimed at producing a high quality *design* (in the hardware sense). Testing and quality control procedures used in manufacturing or maintaining a hardware product are not directly relevant to software quality assurance.

Finding design errors is difficult for both hardware and software engineers, and finding such errors is more difficult as the complexity of the product increases. In general, software products are equivalent to very complex hardware products, so it is no wonder developing high-quality software is difficult and costly.

B. Sources of Software Errors

The basic steps in developing a software system are

1) defining user requirements;
2) deciding what functions and major components a system must provide to meet these requirements;
3) designing and specifying the intended behavior of individual software components;
4) implementing (i.e., coding) software components.

Each of these software development activities is subject to error

1) construction errors—failure of software components, as implemented, to satisfy their specifications;
2) specification errors—failure to accurately specify the intended behavior of a unit of software construction;
3) functional design errors—failure to establish an overall design able to meet identified requirements;
4) requirements errors—failure to identify user needs accurately, including failure to communicate these needs to software designers.

Different quality assurance techniques are required to deal with these errors, since they arise at different stages of software development. In this paper, we will concentrate on design, specification, and construction errors.

Software is always tested before being released for operational use and testing is typically the last activity before release. This gives testing high visibility to developers and users and often leads to an undue reliance on tests as the means of ensuring software quality.

Since tests are performed only after code is written, they are a costly method of detecting errors, especially specification and design errors. Correcting these errors may require discarding some software that has already been written and tested. A cost effective approach to reducing software errors must attempt to prevent and detect errors as soon as possible. Ideally, software design errors are detected during the design phase, before specifications of software components are written, and inconsistencies between specifications and the design are detected before code is written.

In Section II, we will discuss principles and practices underlying the development of software tests, and in Section III, we will discuss an integrated approach to software quality assurance. The integrated approach discusses what quality assurance procedures and principles should be applied at each phase of software development, rather than focusing solely on the development and evaluation of test cases.

II. Software Testing Principles

From an engineering point of view, software testing has traditionally been *ad hoc*, with few principles and no theoretical work indicating how to construct an adequate set of tests or how to measure the adequacy of a set of tests that someone

has constructed. In the last few years, however, some theoretical work has increased our understanding of how to construct tests. We will discuss this recent work to show how the basis for an engineering approach to software testing is developing.

Software testing involves the execution of programs with selected inputs, called test cases. The results of test executions are then used to decide whether the program is operating acceptably.

The most common objective in software testing is to determine whether a program is correct, i.e., whether the program produces specified outputs when presented with permitted inputs. Although correctness may at first seem to be the most important property a program can have, this is by no means the case, particularly for large software systems. (See [20] for a more detailed discussion.) Large programs are often so complex they never completely satisfy their specifications, and yet, they may be quite usable because failures are encountered infrequently in practice, and when they do occur, their impact on a user is acceptably small. Hence correctness is not necessary for a program to be usable and useful. Nor is correctness sufficient. A correct program may satisfy a narrowly drawn specification and yet not be suitable for operational use because in practice, inputs not satisfying the specification are presented to the program and the results of such incorrect usage are unacceptable to the user. If a program is correct with respect to an inadequate specification, its correctness is of little value.

Consequently, although testing for correctness is the most common and best understood testing goal, correctness is by no means the only important property of usable software—reliability, robustness, efficiency, and other properties (see [12]) are also of significant importance. But these properties are less commonly the focus of testing activities.

In designing correctness tests, it is important to keep a few principles in mind.

1) "Black box" tests (i.e., test cases chosen without knowledge of how a program has been implemented) cannot ensure that all correctness errors are detected unless the tests are exhaustive, i.e., consist of all inputs in the program's input domain. Exhaustive tests are almost never practicable, however. Input domains are just too large; often they are not finite.

2) Even "glass box" tests (i.e., tests chosen with full knowledge of how a program has been implemented) are not necessarily adequate to detect all correctness errors. For example, selecting tests so all branch conditions in a program or even all execution paths through a program are exercised will not detect an error if a branch or path that should be present in the program is missing (see [12]).

3) The most effective way to design tests is to hypothesize certain software errors and then to select test cases that will fail if the errors are present. We discuss aspects of this approach below.

Current research approaches for developing correctness tests fall in two catagories—deterministic and probabilistic. Deterministic methods select tests that will fail if certain kinds of errors (and only those kinds of errors) are present in a program. Probabilistic methods provide estimates of the likelihood of undetected errors remaining in a program without ever fully guaranteeing that all errors (of certain kinds) have been eliminated. Probabilistic methods are aimed not so much at defining how to select test data but rather at evaluating the effectiveness of tests in uncovering errors. An ideal software testing method gives both a reliable measure of effectiveness and, if the measure indicates more testing is needed, shows a tester where further testing effort will be profitable.

For example, the deterministic approach is illustrated by Chow's [6] technique for testing communications software protocols and programs implementing them. Such protocols can be described in terms of finite state machines. Chow has shown that given an upper bound on the number of states in the correct finite state machine, a set of test cases can be generated that will detect all errors due to missing states and missing or incorrect state transitions. A program implementing the protocol can be deterministically tested with these inputs to determine if it has realized the intended design, i.e., successful execution implies the implementation is correct. West [31] describes a similar procedure for detecting system deadlocks, potential losses of messages, and other communications protocol errors. West also requires that the protocols be modeled as finite state machines.

Another example of the deterministic approach is the one developed by White and Cohen [32]. They attempt to detect only errors due to incorrectly written branch conditions in a program. They partition the input domain of a program into equivalence classes such that each element of a class causes execution of the same program control flow path. Their testing strategy describes how to select test data to determine if the boundaries defining a class have been shifted from their correct position. For example, given a boundary defined by an ordering relation, e.g., $3X + 2 > Y$, they show why in general three test cases are needed to show whether the boundary has been correctly specified—two test points on the boundary and one off the boundary, e.g., (3, 11), (-1, -1), and (1, 4). A more naive approach to test case selection would assume that two test cases would suffice—one in which the relation was satisfied, and one in which it was not. But Cohen and White's analysis shows why two cases are insufficient to detect errors in the coefficients. Showing how intuitive testing approaches can be inadequate is an important result of deterministic testing research.

The common elements in deterministic testing approaches are 1) their focus on detecting only certain kinds of errors in the absence of other kinds of errors, and 2) their development of methods for selecting test data that is guaranteed to detect these errors if they are present. A productive area for further research is defining test case selection methods for additional classes of errors. Such research is needed to provide a firm theoretical basis to guide software engineering practice.

In constrast to the deterministic approach to testing, in which one attempts to find test cases guaranteed to detect certain kinds of errors, DeMillo *et al.* [8] have developed a probabilistic method for assessing the quality of a test set without knowing precisely what errors the test set is able to detect. Their method requires "mutating" (i.e., modifying) the program to be tested by introducing small changes that are likely to be errors. The original program and the mutated programs are tested using the same set of test cases. The quality of the test set is determined by the number of mutants generated and by how many fail to pass the tests. Ideally, all mutants fail. When mutants are not eliminated by the tests, one must either attempt to find test data for which the mutants will fail or demonstrate that the mutants are equivalent to a correct program.

The principle underlying the mutation approach to testing is "Test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors" [8].

There is no certainty that all errors are discovered if the set of tests eliminates all incorrect mutants. The assumption, however, is that in practice any remaining errors are not important. Research is currently underway to see to what extent this assumption holds and to see what kinds of program mutations are most useful for validating test sets [1].

Software testing is not currently an engineering discipline. The theoretical basis needed for such a discipline simply does not yet exist. But our examples of new testing approaches show that some progress is being made.

While waiting for theory to catch up with the needs of software developers, today's software engineers must apply rules of thumb learned by experience. These rules are well explained by Myers [23], [24], whose books contain what is probably the best currently available collection of good practices for developing software tests.

Perhaps the most important thing to remember about software testing is that although it is necessary, it cannot be the sole means of ensuring software is reliable, robust, and useful. To obtain software having these properties in sufficient measure, one must use appropriate quality assurance techniques at each stage in the software development process. Testing is only one of these stages.

III. An Integrated Approach to Quality Assurance

As noted previously, software development is analogous to the building and testing of a hardware prototype. Accordingly, attempts to assure software quality should attend principally to the design process and should analyze its outputs. Indeed, H. Mills [18] recommends that

> Given double the budget and schedule (to test the sincerity of a requirement for ultrareliability), do not spend the extra on testing–spend it in design and inspection.

In this section, we review the software development process's major activities, namely design, construction, and testing, from the perspective of what proven quality assurance principles and procedures should be applied. Coverage of all the applicable techniques and tools is not possible here. Rather, the intention here is to survey the state of the art as practiced with suitable references supplied for reader follow up.

A. Reducing Design and Specification Errors

Software design can be characterized as allocating requirements to the components of an architecture. This characterization stresses that a design consists of parts (modules) and their interconnections (interfaces) for the purpose of realizing a given set of requirements. Clearly this presupposes that the software requirements are defined and analyzed prior to the design activity. Without first satisfying this important presupposition all subsequent efforts to assure a quality product are, at best, misguided.

Presently, there are quite a few competing "design methodologies" offering a software designer (or a design team) prescriptions of explicit steps to follow as well as a specific, graphic notation for representing the resultant design. We mention here some of the more prominent approaches: structured design [21], [22], [29], [36], the Jackson method [15], the Warnier–Orr approach [25], [30], the structured analysis and design technique (SADT) [9], [27], the requirements engineering validation system (REVS) [2], the systematic design methodology (SDM) [3], [14], the operational software concept [26], [35], higher order software (HOS) [13], and the architecture definition technique (ADT) [4].

Generally by using a top-down strategy, each of these methods develops a software system design as a group of communicating modules. Such top-down strategies proceed from the general to the particular by first defining the context, the function, and the interfaces for any module before specifying the internal details of that module. However, the various methods are fundamentally different in what they regard as important and in how they attack a design problem. Naturally the graphic notation associated with a particular method is well suited to representing "important" system aspects. That is, the very notation for representing a design architecture enforces abstraction by excluding or deferring less important aspects (as judged by a method's creators). Some of the system aspects that have been used by various design methods for creating a "good" design are: control flow, data flow, the input and output data structures, the internal data-base structure, stimulus/response threads through the system, hierarchical decomposition of system functions, system states (or modes), and the transitions between them, dependencies between functions, and major system features that should be easily changeable.

For a software quality assurance viewpoint, there are many advantages to adopting a particular design method and graphic notation for representing a design. Such a standard establishes a basis for training and for project communication. This eases the introduction of new personnel during development and provides a useful "mental framework" for the eventual maintainer. Once selected, design standards enhance the efficacy of all internal design reviews by providing some objective criteria that can be uniformly applied when assessing design "qualities" such as completeness, correctness, and clarity. Moreover, standard design documentation can–and should–be supported by software tools that store the design information for subsequent queries, that automatically produce a standard formatting, and that report on the consistency of the design information entered thus far. Some research and development efforts (based on particular design methods) have extended the design support tools to include simulations or "animations" of the currently stored design. In summary, with respect to using a design method experience indicates that any reasonably systematic strategy is better than none.

Besides design reviews internal to the development team, several formal design reviews for the benefit of senior management, the customer, and an independent quality assurance group are advisable. Established practice now holds two such formal exercises called the preliminary design review (PDR) and the critical design review (CDR). The PDR assesses the proposed design architecture for logical and technical feasibility. All of the software components (both procedural and data) should be functionally specified with interfaces identified in some detail. In addition, a plan scheduling subsequent design, implementation, testing, and integration tasks must be provided. Draft user and maintenance manuals and a proposed acceptance test plan are appropriate. The CDR focuses on implementation and performance feasibility. It typically requires the detailed algorithms, data structures, and interface formats along with memory and execution time budgets for each software elements. Usually satisfying the CDR "action items" is the prerequisite to initiating actual coding. With respect to internal and formal design reviews Glass [11] has said the following.

> Most important, the success or failure is dependent on people. The people who attend must be intelligent, skilled, knowledgeable

in the specific problem area, cooperative, and have the time available to dedicate themselves to the review. Only the interactions of capable people can make a design review successful.

The quality assurance role in a design review is to examine the design for completeness (are all requirements addressed?), for quality (using objective criteria derived from the selected design method), and for adherence to standards.

B. Reducing Construction Errors

Software construction is concerned with the design of module details and then with expressing these details in executable code. Algorithm design, data layout, and access considerations are central to this activity. Already there is a rich body of knowledge about specific algorithms and data structures as well as their relative performance tradeoffs. By demonstrably helping to improve the quality of software construction, two general practices have gained widespread acceptance—structured programming for detailed routine and data design and a higher order language (HOL) for coding programs.

Structured programming involves the hierarchical fabrication of programs and data structures by means of the repeated application of some basic composition rules. For example, in "structured coding" basic statements (such as arithmetic evaluation and assignment) may be combined into compound statements only by using statement sequencing, conditional selection of a group of statements to be executed, or conditional iteration of some statements. That is, the unconditional branch or GO TO statement is excluded from structured coding (sometimes erroneously characterized as GO TO-less programming). These restrictions generally simplify a routine's flow-of-control logic and have demonstrably reduced a significant source of coding errors. In less than a decade, structured programming has gone from a proposed approach [7], [19], [33] to an operational standard for most major software developers.

Coding standards incorporate aspects of structured programming with rules for indentation (to reflect nesting), header comments, identifier names, and the length of routines (to encourage short, functional procedures). A major premise of structured programming is that programs are to be read by people as well as by computers. Proceeding on this premise, quality assurance activity has recently included code reading. The traditional desk checking by the programmer is profitably augmented with a more formal peer code review [10] that is scheduled, conducted using checklists for guidance, and tracked with all the trappings of reports, action items and follow ups.

Using an HOL makes software construction much more reliable and productive than using assembly language. Good optimizing compilers can now produce code that is within 25 percent in both space and execution time of well-crafted assembly language. HOL code is so much easier to produce, checkout, maintain, and modify that it should be used for all but the most stringently constrained target environments. Indeed, Fred Brooks claims, "I cannot easily conceive a programming system I would build in assembly language" [5]. Interactive debugging in the symbolic terms of the HOL source program substantially speeds routine check out. Modern typed HOL's (such as Pascal [34]) catch many data interface and misuse errors during compilation. In some cases, detailed design can be better represented in a suitable source HOL (because of the automatic check out and cross referencing) than in flowcharts or in some program design language (PDL).

C. Extending Testing Principles to Software Systems

The burden of producing detailed design documentation and of unit testing various routines often obscures what the essential product of software development is. The real goal should be to place a system (including people, machines, and software) into satisfactory operation. Consequently system and software integration are prominently placed on the critical path. To better reduce and control the associated risk, software integration and testing efforts should be distributed over as much of the coding effort as possible. Indeed, a more accurate measure of current project development status is "what percentage of the total number of software modules identified during architectural design have been coded and integrated into an operational subsystem?" Such a measure is far superior to the more frequently used "how many lines have been coded?" or "how many modules have been unit tested?" Clearly a percentage-of-modules-integrated measure dictates using some variant of a top-down implementation and testing sequence [17]. A significant strategy is to identify some operational subsets (or "builds") of the intended system and to develop and deliver in succession these subsets [28]. This approach provides an early check out of the major software and user interfaces while helping to identify some important but nontechnical potential problems (e.g., user training, delivery format and mechanism, coordination of multiple contractors, etc.). In fact, testing should drive development in the sense that plans for testing software subsets should determine the implementation sequence.

Testing can be regarded as gathering information on the software's reliability. Thus viewed, test requirements, plans, and procedures as well as design reviews are as much a part of testing as is exercising software on a machine. It is beneficial to have a separate project test or quality assurance team address these issues. For example, a test procedure must clearly define the series of actions required to verify that a product meets its requirements. These actions may include

1) configuring—arranging the software, hardware, people, and logistics for the test;
2) conditioning—bringing the system to the initial state for testing;
3) introducing data—entering either "live" or simulated data;
4) starting the test—initiating and synchronizing where necessary;
5) collecting data—gathering snapshot, frequency, and resource utilization information;
6) displaying results—selecting and formatting results as directed;
7) analyzing results—comparing actual to expected.

A variety of software tools assist in conducting test procedures. These include environment simulators, test data generators, and test coverage analyzers.

In summary, software development and test is fairly analogous to the development and testing of a hardware prototype; design errors predominate. This insight helps in identifying some major sources of software errors. A serious software quality assurance effort that focuses on these error sources must be active from a project's inception. The issues of requirements testability, traceability, and cost must be resolved early. Design reviews must be conducted for the purpose of finding design errors sooner rather than later. Proven methods for designing, programming, making the evolving product

visible, and controlling different system configurations can be standardized and then supported by a suitable software project library with associated procedures. Of course, execution testing remains the principle means for determining the current status of a software product.

REFERENCES

[1] A. T. Acree, R. A. DeMillo, T. J. Budd, R. J. Lipton, and F. G. Sayward, "Mutation analysis," NTIS Rep. ADA-076575, Sept. 1979.

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

[3] R. Andreu, A systematic approach to the design and structuring of complex software systems, Ph.D. dissertation, Sloan School of Management, Massachusetts Institute of Technology, 1978.

[4] C. Bachman and J. Bouvard, "Architecture definition technique: its objectives, theory, process, facilities and practice," in *Proc. ACM SIGFIDEF Data Description Access and Control*, pp. 257-305, 1972.

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

[6] T. S. Chow, "Testing software design modeled by finite-state machines," *IEEE Trans. Software Eng.*, vol. 4, pp. 278-186, May 1978.

[7] O.-J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, *Structured Programming*. New York: Academic Press, 1972.

[8] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Comput.*, vol. 4, pp. 34-43, Apr. 1978.

[9] M. Dickover, C. McGowan, and D. T. Ross, "Software design using SADT," in *Proc. Nat. ACM Conf.*, pp. 125-137, 1977.

[10] M. Fagen, "Design and code inspections to reduce errors in program development," *IBM Syst. J.*, vol. 15, pp. 182-211, 1976.

[11] R. Glass, *Software Reliability Guidebook*. Englewood Cliffs, NJ: Prentice-Hall, 1979.

[12] J. B. Goodenough, "A survey of program testing issues," in *Research Directions in Software Technology*, P. Wegner, Ed. Cambridge, MA: M.I.T. Press, 1979, pp. 316-340.

[13] M. Hamilton and S. Zeldin, "Higher order software—A methodology for defining software," *IEEE Trans. Software Eng.*, vol. 2, pp. 9-32, June 1976.

[14] S. Huff and S. Madnick, An extended model for a systematic approach to the design of complex systems, NTIS Rep. ADA-058565, 1978.

[15] M. A. Jackson, *Principles of Program Design*. New York: Academic Press, 1975.

[16] M. M. Lehman, "Programs, programming, and the software life cycle, this issue, pp. 1061-1076.

[17] C. L. McGowan and J. R. Kelly, *Top-Down Structured Programming Techniques*. New York: Van Nostrand, 1975.

[18] H. Mills, "Software development" in *Research Directions in Software Technology*, P. Wegner Ed. Cambridge, MA: M.I.T. Press, pp. 87-105.

[19] —, "Top-down programming in large systems," in *Debugging Techniques in Large Systems*, R. Rustin, Ed. Englewood Cliffs, NJ: Prentice-Hall, 1971, pp. 41-55.

[20] J. D. Musa, "The measurement and management of software reliability," this issue, pp. 1131-1143.

[21] G. J. Myers, *Reliable Software Through Composite Design*. New York: Van Nostrand, 1975.

[22] —, *Composite/Structured Design*. New York: Van Nostrand, 1978.

[23] —, *Software Reliability: Principles and Practices*. New York, Wiley, 1976.

[24] —, *The Art of Software Testing*. New York: Wiley, 1979.

[25] K. Orr, *Structured Systems Development*. New York: Yourdon, Inc., 1977.

[26] J. Rodriguez and S. Greenspan, "Directed flowgraphs: The basis of a specification and construction methodology for real-time systems," *J. Syst. Software*, vol. 1, pp. 19-27, Jan. 1979.

[27] D. Ross, "Structured analysis: A language for communicating ideas," *IEEE Trans. Software Eng.*, vol. 3, pp. 16-33, Jan. 1977.

[28] D. Schultz, "A case study in system integration using the build approach," in *Proc. Annu. ACM Conf.*, pp. 143-151, Oct. 1979.

[29] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Syst. J.*, vol. 13, pp. 114-139, 1974.

[30] J. Warnier, *Logical Construction of Programs*. Leiden, Germany: Stenfert Kroese, 1974.

[31] C. H. West, "General technique for communication protocol validation," *IBM J. Res. Develop.*, vol. 22, pp. 393-404, 1978.

[32] L. J. White and E. I. Cohen, "A domain strategy for computer program testing," *IEEE Trans. Software Eng.*, vol. 6, pp. 247-257, May 1980.

[33] N. Wirth, "Program development by stepwise refinement," *Commun. Ass. Comput. Mach.*, vol. 14, pp. 221-227, Apr. 1971.

[34] —, "The programming language Pascal," *Acta Informatica*, vol. 1, pp. 35-63, 1972.

[35] K. Wong and J. Engelland, "Operational software concept: A new approach to avionics software," in *Proc. AIAA Digital Avionics System Conf.*, 1975.

[36] E. Yourdon and L. Constantine, *Structured Design*. New York: Yourdon Inc., 1975.