THE JOURNAL OF
COMPUTER INFORMATION SYSTEMS
A Refereed Publication

THE JOURNAL OF COMPUTER INFORMATION SYSTEMS (ISSN 0887-4417 USPS 279-400) is the official publication of the Association for Computer Educators and is published quarterly at 217 College of Business, Oklahoma State University, Stillwater, Oklahoma. Articles in the JOURNAL are indexed in the Business Education Index and University Microfilms. Reprints of articles are available from University Microfilms International, 300 North Zeeb Road, Ann Arbor, Michigan 48106. The Association for Computer Educators does not assume responsibility for views or opinions expressed by contributors to the JOURNAL. The Association for Computer Educators executive office is located at James madison University, School of Business Administration, Harrisonburg, VA 22807. A limited number of back issues of the JOURNAL are available for $12.00 per issue. Requests should be mailed to THE JOURNAL OF COMPUTER INFORMATION SYSTEMS, Dr. Jeretta Horn Nord, Editor, College of Business Administration, Oklahoma State University, Stillwater, OK 74078. Subscriptions (quarterly October through July) are included in ACE individual membership dues of $25 per year, of which $20 is for the subscription. Nonmember subscriptions are available at $25 annually to libraries and institutions. Foreign subscriptions are $35 per year with the exception of Canada and Mexico which are $30 annually. POSTMASTER: Send address changes to The Association for Computer Educators, James Madison University, School of Business Administration, Harrisonburg, VA 22907.

THE JOURNAL OF COMPUTER INFORMATION SYSTEMS
Volume XXX, Number 1 • Fall 1989

CONTENTS

SOFTWARE TESTING TECHNIQUES IN INFORMATION SYSTEMS CURRICULA

by

ELDON Y. LI
California Polytechnic State University
San Luis Obispo, California

ABSTRACT

Software testing is an important part of information systems (IS) development process. To achieve effectiveness in software testing, the participating IS professionals must apply software testing techniques. A review of the current IS curriculum models reveals that specific pedagogical guidelines are not available for instructing software testing techniques. This paper discusses the importance of software testing to IS development and maintenance, reviews the existing software testing techniques, and provides a pedagogical guideline for instructing software testing techniques in IS curricula.

INTRODUCTION

Software quality is one of the major factors influencing the quality of the information systems (IS) in organizations. It is therefore necessary for every completed software product to pass a series of quality tests before it is formally released to its users. In this sense, software testing becomes a mandatory process in the life cycle of a software project. Any IS graduate who will participate in a software project must be ready to participate in both the high-level testing activities (such as the requirements-definition walkthrough, external system design, test planning, black-box test-case design, system testing, and acceptance testing) and the low-level testing activities (such as internal system design, specifications walkthrough, code review, white-box test-case design, and numerous test executions). In order to perform software testing effectively, the IS graduate is required to have the knowledge of software testing techniques. These techniques each provide a structured approach to design test cases and data for testing the quality of a software product. Therefore, they are of vital importance to every practicing IS professional as the other structured techniques such as structured analysis, structured design, and structured programming are.

A sound information systems curriculum should equip its students with both technical and organizational skills in communications (both oral and written), analysis, design, programming, testing, documentation, and management because most of the entry-level jobs opened for college IS graduates require these skills. A review of the current ACM (20) and DPMA (7) curriculum models reveals that specific pedagogical guidelines are available for most of these skills except software testing. Although software testing activities such as walkthrough and review, unit and integration testing, regression testing, and test cases/data design are recommended as the required topics in the systems development courses (such as IS8 [20], CIS/86-3, and CIS/86-4 [7]), neither model provides adequate references for further reading, nor do they indicate what techniques of software testing should be imparted to the IS students. This paper rectifies these deficiencies by providing a guideline for instructing software testing techniques in IS curricula. The existing software testing techniques are reviewed and a set of effective techniques is identified. This set of techniques is then applied to a programming assignment to demonstrate a structured process of software testing. This structured process can serve as a pedagogical guideline for classroom instruction.

SOFTWARE TESTING TECHNIQUES

Software testing techniques can be classified into two groups: the "black-box" and the "white-box" techniques (19). The differences between these two groups of techniques lie in their methods of test-case design. The black-box techniques derive the test cases from the requirements definition or the external (design) specification, while the white-box techniques from the program logic in the source code or internal design specification. The former techniques focus on the functions of the program/system being tested while the latter on the structure. Therefore they are also known respectively as the functional and the structural techniques(1). The test-case design methods of these two groups of techniques are briefly described below.

Black-Box Test-Case Design Techniques

• **Equivalence Partitioning** – requires that the input conditions of the base document (either the requirements definition or the external specification) be partitioned into one or more valid and invalid equivalence classes. When deriving test cases, it requires that all valid input classes be covered before covering any invalid class. When covering the valid input classes, each test case should be derived to cover as many uncovered valid classes as possible. Once all the valid input classes have been covered, each test case should be derived to cover only one uncovered invalid input classes at a time (19).

• **Boundary Coverage** – requires that the input conditions on and adjacent to the boundary of the input equivalence class be tested and that the result space (i.e., the normal-end and the abnormal-end output equivalence classes) be considered and tested as well (12, 19). This method is very useful in generating test data for each test case.

• **Cause-Effect Graphing** – requires that the specifications be divided into smaller workable pieces, that the valid and invalid input conditions (causes) as well as the normal-end and the abnormal-end output conditions (effects) be identified for each workable piece, and that the semantic content of the specifications be analyzed and transformed into a Boolean graph linking the causes and the effects. The graph is then converted into a limited-entry decision table that meets all environmental constraints, and each column in the table represents a test case (8, 9, 19). Cause-effect graphing explores all combinations of input conditions within a workable piece of the specifications while boundary coverage and equivalence partitioning do not.

• **Error Guessing** – requires that a list of possible errors or error-prone situations be enumerated and that test cases be derived based on the list (19). Unlike the boundary coverage technique, error guessing is largely an intuitive (16) and ad hoc process. It relies heavily on the tester's experience. Many test cases derived from this technique are found to overlap those from equivalence partitioning and boundary coverage (1).

White-Box Test-Case Design Techniques

• **Statement Coverage** – requires that every statement in the program be executed at least once (16, 19).

• **Decision Coverage** – also called "branch coverage", requires that every true/false branch be traversed at least once and that every statement be executed at least once (16, 18). Apparently, if a program has single entry and single exit, covering every branch implies that every statement will be executed at least once.

• **Condition Coverage** – requires that every condition in a decision take on its true and false outcomes at least once and that every statement be executed at least once (19).

• **Decision/Condition Coverage** – it is the potpourri of the above three techniques. It requires that every condition in a decision take on its true and false outcomes at least once, that each decision take on every possible true/false branch at least once, and that every statement be executed at least once (19).

• **Multiple-Condition Coverage** – is an extension of the decision/condition coverage. It further requires that every possible combination of condition outcomes within each decision be invoked at least once (19). Obviously, this method is superior to the above four techniques.

• **Complexity-Based Coverage** – uses the cyclomatic number in the literature of graph theory (2, 4, 10) to determine the minimal set of required test cases and provides a structured procedure for deriving the test cases directly from the control-flow graph of the intended program. The cyclomatic number of a program equals one plus the number of conditions in the program (15). The program under test must have a single entry and a single exit. The derived test cases functionally meet the criteria required by the multiple-condition coverage. Complexity-based coverage is superior to the multiple-condition coverage because the former further explores possible combinations of condition outcomes between any two consecutive decisions.

RECOMMENDED SOFTWARE TESTING TECHNIQUES

It is obvious that among the six white-box testing techniques, the complexity-based coverage is the best because it encompasses the other five white-box techniques and further covers possible combinations of condition outcomes between any two consecutive decisions. It is not only easy to apply but also enforces one of the structured programming principles -- any program module, be it large or small, must have a single entry and a single exit (17). As to the black-box techniques, we do not recommend error guessing for classroom training not because it is unimportant but because it provides no guideline for deriving test cases. However, one should know that error guessing, like boundary coverage, can help identifying invalid input conditions during equivalence partitioning, cause-effect graphing, or even walkthrough and review processes. It is also very useful during the debugging stage since debugging relies heavily on the programmer's experience.

Among the other three black-box techniques, boundary coverage is a required supplement to all other test-case design techniques because none of the latter techniques fully test the boundary of each input condition as the boundary coverage does. Between the remaining two black-box techniques, cause-effect graphing is superior to equivalence partitioning because it further explores different combinations of input conditions from the equivalence classes. However, drawing the cause-effect graph for a small problem might be easy but it becomes unwieldy quickly as the problem size grows (21). For cause-effect graphing to be effective, it must be automated. Since there is no commercial tool available for cause-effect graphing today, we do not recommend the inclusion of cause-effect graphing in the IS curriculum. Examples of cause-effect graphing can be found in Elmendorf (8, 9) and Myers (19).

In summary, three out of ten existing software testing techniques are recommended to be instructed in an IS curriculum -- most likely in the systems development courses. They are 1) the equivalence partitioning, 2) the boundary coverage, and 3) the complexity-based coverage techniques. All three techniques can be applied not only to the computer-based testing processes such as regression testing, unit and integration testing, system and acceptance testing, but also to the manual testing processes such as desk-checking, walkthrough, and review. Since each technique has its own weaknesses, they should not be used in isolation, but rather they should supplement one another. The following is an example demonstrating how to apply these techniques to program testing from an IS professional's perspective.

AN EXAMPLE

Assuming that an IS professional is assigned a programming project with the following requirements definition:

"The program accepts three integer values from the keyboard. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral." (Adopted from Page 1 of Myers [19])

Now, to accomplish the project, the IS professional should perform the following steps:

(1) derive a set of test cases using the equivalence partitioning technique,
(2) develop a program internal (logic) specification using pseudocode,
(3) draw a control-flow graph to represent the entire program,
(4) derive a set of test cases using the complexity-based coverage technique,
(5) consolidate the test cases obtained in Steps (1) and (4).
(6) design test data for each test case using the boundary coverage technique,
(7) translate the pseudocode into program source code,
(8) conduct actual testing one test-case item at a time using the test data,
(9) repeated the above procedure if necessary until all the test results are identical to the expected results.

The first six steps listed above are demonstrated in details as follows:

Step 1: Apply the equivalence partitioning technique to the requirements definition to derive the test cases. The semantic content of the requirements definition was analyzed and the keywords were underlined as follows:

"The program accepts three integer values from the keyboard. The three values are interpreted as representing the lengths of the sides of a triangle. The program prints a message that states whether the triangle is scalene, isosceles, or equilateral."

The input keywords are *"accept," "three integers," "keyboard," "lengths of sides (of a triangle),"* and *"triangle."* Among these keywords, the words *"three integers"* and *"lengths of sides"* are strictly data-related; the words *"accept"* and *"keyboard"* are strictly function-related; and the word *"triangle"* is both data-related and function-related. Our focus is on the three data-related keywords, *"three integers," "lengths of sides,"* and *"triangle."* Based on these keywords, the possible valid and invalid input conditions and their corresponding expected output conditions are enumerated in Table 1. Each expected output condition represents a unique test case for the intended program.

TABLE 1
Test Cases Derived from Equivalence Partitioning

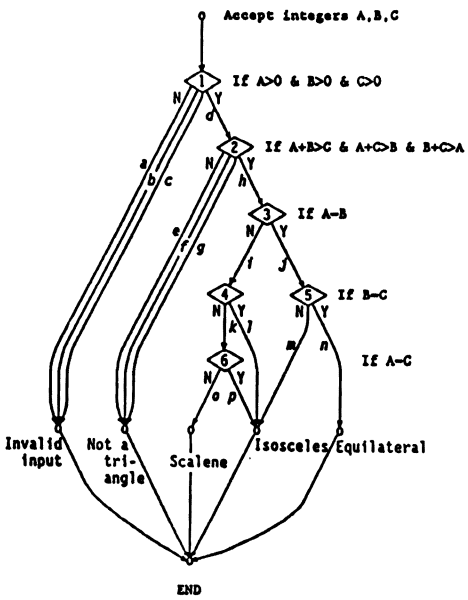
Input Equivalence Classes	*Test Case I.D. (Input Equivalence Classes Being Covered)*
Valid:	
Three Integers:	
1. A is an integer	1. a triangle (1-9)
2. B is an integer	
3. C is an integer	
Lengths of Sides:	
4. A>0	
5. B>0	
6. C>0	
Triangle:	
7. A+B>C	
8. A+C>B	
9. B+C>A	
Invalid:	
10. A<1 & A is an integer	2. invalid integer A (10)
11. B<1 & B is an integer	3. invalid integer B (11)
12. C<1 & C is an integer	4. invalid integer C (12)
13. A is not an integer	5. non-integer A (13)
14. B is not an integer	6. non-integer B (14)
15. C is not an integer	7. non-integer C (15)
16. A+B≤C	8. not a triangle (16)
17. A+C≤B	9. not a triangle (17)
18. B+C≤A	10. not a triangle (18)

Step 2: One possible set of pseudocode for this program is written below. Note that pseudocoding in the program internal specification emphasizes not the efficiency (structure) but the effectiveness (functions) of the desired program. Our pseudocode here may not be efficient but it is certainly effective.

```
PROGRAM TRIANGLE (A,B,C)
ACCEPT integers A,B,C from the keyboard.
IF A>0 AND B>0 AND C>0 THEN
    IF A+B>C AND A+C>B AND B+C>A THEN
        IF A=B THEN
            IF B=C THEN PRINT "equilateral,"
            ELSE PRINT "isosceles,"
        ELSE IF B=C THEN PRINT "isosceles,"
            ELSE IF A=C THEN PRINT "isosceles,"
                ELSE PRINT "scalene,"
    ELSE PRINT "not a triangle,"
ELSE PRINT "invalid input."
END of program.
```

Step 3. Figure 1 shows the control-flow graph representing the program pseudocode. The graph is drawn by McCabe's (15) convention which uses multiple branches to represent the true/false outcomes of a compound decision (i.e., a decision with AND or OR operators). All the decisions and outcome branches are labeled to facilitate test paths identification.

FIGURE 1
A Control-Flow Graph for the Triangle Program



Step 4: This step is to develop a set of test cases using the complexity-based coverage technique introduced by Thomas E. McCabe (15) which allows the tester to find all independent paths directly from the control-flow graph of a program. Each path found represents a test case for testing the program. McCabe's method as it applies to the control-flow graph of Figure 1 is summarized below along with the author's notation.

Procedure for Complexity-Based Coverage:

(1) Pick a functional "baseline" path through the program which represents a legitimate function and not just an error exit. The key is to pick a representative function provided in the program as opposed to an error path that results in an error message or recovery procedure. For example, path *1d2h3i4k6p* is a possible baseline. Note that our path expression is somewhat different than that of McCabe (15) in which the decision number does not appear.

(2) Identify the second path by locating the first decision on the baseline and flipping its outcome while simultaneously holding the maximum number of the original baseline decisions unchanged. If the decision has multiple conditions, each condition should be flipped one at a time. This process is likely to produce a second path which is minimally different from the baseline path. The result yields three paths: *~1a*, *~1b*, and *~1c*. We use the symbol "~" to indicate that the decision behind the symbol has been flipped.

(3) Set back the first decision to its original value before the flipping, identify the second decision in the baseline path, and flip its outcome while holding all other decisions to their baseline values. This process, likewise, should produce a third path which is minimally different than the baseline path. The result yields another three paths: *1d~2e*, *1d~2f*, and *1d~2g*.

(4) Repeat this procedure until one has gone through every decision on the baseline and has flipped it from the baseline value while holding the other decisions to their original baseline values. After flipping the third decision, we have the path *1d2h~3j5m*. Flipping the fourth and the sixth

decisions yields in sequence the paths *1d2h3i~41* and *1d2h3i4k~6o*.

(5) Repeat the above procedure for any unflipped decision which is not on the baseline. Once all the decisions have been flipped, the process is then completed. In our case, we must flip the fifth decision encountered in Step (4). Flipping the fifth decision yields the path *1d2h~3j~5n*.

Table 2 shows the eleven paths found by the complexity-based coverage technique and their corresponding test-case numbers of Table 1. Notice that the number of test cases derived from this procedure (which is 11) always equals the cyclomatic number of the program which is one plus the number of decision conditions in the program (which is 10).

TABLE 2
Test Cases Derived from the Complexity-Based Coverage

Case I.D.	Test Paths (Cases) Derived from the Complexity-Based Coverage	Test Case I.D. in Table 1
1.	1d2h3i4k6p (Baseline)	1*
2.	~1a	2
3.	~1b	3
4.	~1c	4
5.	1d~2e	8
6.	1d~2f	9
7.	1d~2g	10
8.	1d2h~3j5m	1*
9.	1d2h3i~4l	1*
10.	1d2h3i4k~6o	1*
11.	1d2h~3j~5n	1*
12.	**	5**
13.	**	6**
14.	**	7**

* The input conditions identified by the equivalence partitioning technique does not require the input conditions for different types of triangle. In contrast, the pseudocode as well as the control-flow graph further considered the possible types (i.e., outcomes) of a triangle.

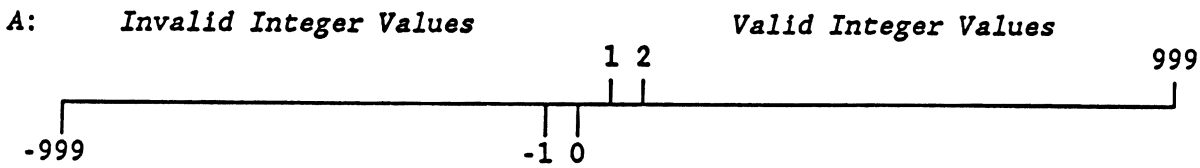
**This test case does not have a corresponding test path because the pseudocode as well as the control-flow graph assumes that the input will be of integer format and that the format will be checked by the system. In contrast, the requirements definition makes no such assumption.

Step 5. The cross-reference in Table 2 reveals that the test paths/cases derived by the complexity-based coverage technique may not perfectly match those derived by the equivalence partitioning. Because that the pseudocode was written based on the assumption that the system will check the input format and only accept integer input, the complexity-based coverage technique did not identify the test cases covering non-integer input conditions. On the contrary, equivalence partitioning did identify the test cases covering non-integer input conditions, but it did not derive the test cases examining different types of triangle as the complexity-based coverage did. Since our objective is to derive and use as many independent test cases as possible, we shall consolidate the two sets of test cases and use all the fourteen test cases listed in Table 2 to derive test data. One word of caution is that for the complexity-based method to be effective, the target program or pseudocode must be coded according to the structured-programming principles (e.g., single entry and exit, no unconditional GOTO branch, use of structured constructs, etc. [3, 5, 6, 17]).

Step 6: Equivalence partitioning and complexity-based coverage techniques are best for deriving possible test cases, but when it comes down to generating test data, both techniques must be supplemented by the boundary coverage technique. For example, one of our valid input equivalence classes is delineated by "A>0 & A is an integer," the lower boundary values of this input condition are A=1, A=1+*e* and

A=1-*e*, where *e* is the minimum significant unit of measure which is "1." Therefore, we generate A=1, A=2, and A=0 one at a time as the test data. If the A is a real number, we generate A=1, A=1.001, and A=.999. On the other hand, the upper boundary is a very large integer number, say A=999. The invalid input equivalence class of A being an integer is then "A≤0 & A is an integer." The upper boundary values of this invalid class are A=0, A=1, and A=-1, while the lower boundary value is A=-.999. However, the value of A=-999 is redundant since any negative values of integer A will be rejected by the program/system and the value A=-1 already covered this case. The value of A=-1 is preferred to A=-999 because the former is near the boundary between the valid and invalid input classes. The boundary values of the input integer A are indicated in Figure 2. By the same token, the test-data values of B and C are similarly assigned.

FIGURE 2
The Boundary Values of the Input Integer A



The other input condition is A+B>C which has two invalid boundary conditions: A+B=C and A+B<C. Therefore, we create two sets of test data: {A=1, B=1, C=2} and {A=1, B=1, C=3}. The test data for the input conditions A+C>B and B+C>A are derived as expected.

With respect to the boundary of the output space, it was found that each expected unique type of triangle does not have a matching input equivalence class. However, this problem was overcome by the test cases derived from the complexity-based coverage. The test data for each test case along with its expected test outcome are enumerated on the last two columns of Table 3. These test data completely cover the boundaries of the output space.

Note that the use of boundary coverage method is only limited by one's imagination. For example, it can be applied to the following cases:

(1) A program processes several arrays. Test both the upper and the lower boundary subscripts of each array (13).
(2) A program updates a file. Process the file without any change, then with a change of the first record, then a change of the last record, finally, a change of a record which does not exist in the file.
(3) A main program which calls four independent modules will display a menu of module numbers, names, and functional descriptions, and prompts for the user's selection of one of the module number, 1 through 4. Test the main program by selecting 0, 1, 4, and 5.
(4) A program contains a DO loop with an exit condition. Test the loop with 0 entry, 1 entry, and 2 entries. This coverage method is known as the "boundary-interior" path testing procedure (11).

Steps 7, 8 and 9: Finally, the IS professional will translate the pseudocode into program source code, and then test the source code by executing it with one set of test data at a time. To complete the testing of source code, all 23 test-case items listed in Table 3 must be executed. If any major error was found, the error should be removed before the testing process is continued. Repeat the above process to re-design the test cases/data or re-code the program and, to re-test the program until all the test results are identical to the expected results.

DISCUSSION

The above discussion focused on those testing techniques which are essential to the IS professionals in testing their IS software. Other techniques such as proof of correctness, simulation, symbolic execution, among others (1) are not necessary to the IS professionals and thus were not discussed in this paper. Although the example used in this paper may seem trivial, the basic principles of the testing techniques and the testing process demonstrated in the previous section can be applied to a program/system of any size (be it large or small) and to any level of computer-based testing as well as manual desk checking.

Since the process demonstrated above is highly structured and straightforward, it is pedagogically feasible for classroom instruction and practices. We have imparted this process to our students in the system design and implementation course

TABLE 3
Final Test Cases and Test Data Generated by the Boundary Coverage

Test Item I.D.	Test Paths Derived by the Complexity-Based Method	Test Case I.D. in Table 1	Expected Test Outcomes	Test Data Derived by Boundary Coverage for Each Test Case***		
1	~1a	2	Invalid A	A = 0	B = *	C = *
2				A = .1	B = *	C = *
3	~1b	3	Invalid B	A = 1	B = 0	C = *
4				A = 1	B = -.1	C = *
5	~1c	4	Invalid C	A = 1	B = 1	C = 0
6				A = 1	B = 1	C = -1
7	**	5	Non-integer A	A = 1.01	B = *	C = *
8				A = .999	B = *	C = *
9	**	6	Non-integer B	A = 1	B = 1.01	C = *
10				A = 1	B = .999	C = *
11	**	7	Non-integer C	A = 1	B = 1	C = 1.01
12				A = 1	B = 1	C = .999
13	1d~2e	8	Not a triangle	A = 1	B = 1	C = 2
14				A = 1	B = 1	C = 999
15	1d~2f	9	Not a triangle	A = 1	B = 2	C = 1
16				A = 1	B = 999	C = 1
17	1d~2g	10	Not a triangle	A = 2	B = 1	C = 1
18				A = 999	B = 1	C = 1
19	1d2h3i4k6p	1	Isosceles	A = 2	B = 1	C = 2
20	1d2h~3j~5n	1	Equilateral	A = 1	B = 1	C = 1
21	1d2h~3j5m	1	Isosceles	A = 999	B = 999	C = 1
22	1d2h3i~4l	1	Isosceles	A = 1	B = 999	C = 999
23	1d2h3i4k~6o	1	Scalene	A = 2	B = 3	C = 4

* This entry can be of any value.
** No corresponding test case is generated because the integer format is assumed to be checked by the system.
*** Without boundary-value analysis the data may not be the same and the second set of test data for each invalid input condition will not be generated.

and received overwhelming, positive feedback from those who took the course, Our experience indicates that before learning the three recommended testing techniques, each student was using exclusively the error guessing technique -- which is hardly a technique -- to derive test cases and data. After experiencing the above testing process for two or three times, every one of them eventually became an effective test-case designer.

SUMMARY

This paper discusses the importance of software testing to IS development and maintenance, reviews the existing software testing techniques, and recommends a set of effective techniques to be included in IS curricula. The techniques recommended include 1) equivalence partitioning, 2) boundary coverage, and 3) complexity-based coverage. These three techniques were applied to the same programming example to demonstrate a realistic, structured testing process. The demonstrated testing process is recommended as a pedagogical guideline for instructing software testing in a classroom setting.

REFERENCES

1. Adrion, W.R., M.A. Branstad, and J.C. Cherniavsky. "Validation, verification, and testing of computer software," **ACM Computing Surveys**, 14, 2 (June 1982), pp. 159-192.
2. Berge, C. **Graphs and Hypergraphs**, Amsterdam, The Netherlands: North-Holland, 1973, 15-17.
3. Bohm, C., and G. Jacopini. "Flow diagrams, Turing machines and languages with only two formation rules," **Communications of the ACM**, 9, 5 (May 1966), 366-371.
4. Deo, N. **Graph Theory with Applications to Engineering and Computer Science**, Englewood Cliffs, NJ: Prentice-Hall, 1974, 55-58.
5. Dijkstra, E.W. "Go To statement considered harmful," **Communications of the ACM**, 11, 3 (March 1968), 147-148.
6. Dijkstra, E.W. "Structured programming," **Software Engineering Techniques**, Report on a Conference sponsored by the NATO Science Committee, Rome, Italy (April 1970), 84-88.
7. DPMA. **The DPMA Model Curriculum for Undergraduate Computer Information Systems, 2nd Edition**, Data Processing Management Association, Park Ridge, IL, July 1986.
8. Elmendorf, W.R. "Cause-effect graphs in functional testing," TR-OO.2487, IBM System Development Division, Poughkeepsie, New York, 1973.
9. Elmendorf, W.R. "Functional analysis using cause-effect graphs," **Proceedings of SHARE XLIII**, New York, (1974), 567-577.
10. Harary, F. **Graph Theory**, Reading, MA: Addison-Wesley, 1969, 37-40.
11. Howden, W.E. "Methodology for the generation of program test data," **IEEE Transactions on Computers**, C-24, 5 (May 1975), 554-559.
12. Howden, W.E. "A survey of dynamic analysis methods," In Miller, E., and Howden, W.E. (eds.) **Tutorial: Software testing and validation techniques**, New York: IEEE Computer Society, 1981, 209-231.
13. Kernighan, B.W., and P.J. Plauger. **The Elements of Programming Style**, New York: McGraw-Hill, 1974, 61-62 & 89.
14. McCabe, T.J. "A complexity measure," **IEEE Transactions on Software Engineering**, SE-2, 4 (April 1976), 308-320.
15. McCabe, T.J. (ed.) **Structured Testing**, Silver Spring, MD.: IEEE Computer Society Press, 1983, 19-47.
16. Miller, E.F., Jr. "Program testing: Art meets theory," **Computer**, 10, 7 (July 1977), 42-51.
17. Mills, H.D. "Mathematical foundations for structured programming," FSC 72-6012, IBM Federal System Division, Gaithersburg, MD, 1972.
18. Myers, G.J. **Software Reliability**, New York: Wiley-Interscience, 1976, 201-206.
19. Myers, G.J. **The Art of Software Testing**, New York: Wiley-Interscience, 1979, vii, 1-11, & 36-76.
20. Nunamaker, J.F., J.D. Couger, and G.B. Davis, (eds.) "Information systems curriculum recommendations for the 80s: Undergraduate and graduate programs," **Communications of the ACM**, 25, 11 (November 1982), 781-805.
21. Ould, M.A., and C. Unwin. **Testing in Software Development**, Cambridge, Great Britain: Cambridge University Press, 1986, 80-99.