1988

SOFTWARE TESTING TECHNIQUES FOR THE INFORMATION SYSTEMS PROFESSIONAL: A CURRICULUM PERSPECTIVE

Eldon Y. Li
California Polytechnic State University

SOFTWARE TESTING TECHNIQUES FOR THE INFORMATION SYSTEMS PROFESSIONAL: A CURRICULUM PERSPECTIVE

Eldon Y. Li
Management Information Systems
California Polytechnic State University

ABSTRACT

A sound information systems (IS) 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 IS curriculum models reveals that specific pedagogical guidelines are available for most of these skills except software testing. Software testing is an important part the of IS development process. To achieve effectiveness in software testing, the participating IS professionals must apply 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.

1. INTRODUCTION

Information systems (IS) curricula have undergone several changes in the past two decades. Each change was invariably due to the changing demand of the IS job market. It is generally agreed in today's computer industry that the demand for personnel having a combination of technical and organizational skills is much greater than the demand for having either type of skills alone, and that the shortage of personnel with balanced skills is acute (Nunamaker 1981). Due to this shortage, more and more computer science (CS) degree programs encourage their students to equip themselves with organizational skills by taking more management courses (Ardis 1987; Freeman 1987). One CS degree program--*software engineering* (SE)--has been shaped in particular to meet the current job demand (IEEE 1982). Today, SE and IS curricula typically share such topics as system life cycle management, system development project, requirement analysis, systems design, structured programming, management and communication sciences. Nevertheless, the former curricula emphasize technical, computer-related skills while the latter emphasize organizational, system-related skills. As the SE students equip themselves with more and more organizational skills, our IS graduates will be facing intense competition on the job market. Now is the time for us to increase the competition leverage of our IS students by familiarizing them with a set of effective testing techniques.

Viewed in another perspective, computer software is one of the major components of information systems (IS) in an organization. Its quality has a direct impact on the quality of the information systems containing it and, in turn, on the performance of the organization operating the information systems. Therefore, assuring software quality is an essential function of an IS department and the program of software quality assurance (SQA) is vital to the success of IS development and maintenance (Pfau 1978; Stamm 1981; Gustafson and Kerr 1982). Typically, an SQA program contains many processes, e.g., configuration management, testing, corrective action, documentation, reviews and audits, among others (Knight 1977). Among these SQA processes, software testing is the most technical process--a topic over which most IS professionals are not enthusiastic--whose main purpose is to not to improve the quality of a software product but to detect the needs for improvement in a software product. The effort of software testing usually accounts for 30 to 60 percent of the total project effort depending on the size of the software project. Therefore, software testing is a critical process in an IS software project.

From the programmer's productivity standpoint, software testing and software coding should go hand-in-hand. That is, one who performs software coding should test his/her own code before someone else does it. This principle of practice in effect encourages programmers to design their programs for testability and in turn increases their productivity. In addition to the programmers, other IS personnel such as systems analysts, information analysts, and project managers who participate in the project must also participate in different levels of software testing activities--such as integration tests, system test, and acceptance test. Although they may not actually conduct all the tests in which they participate, they need to know how to test them. It is clear that unless our IS graduates always stay in positions which do not require software coding at all--a situation which is very unlikely--software testing will be an important skill of our IS graduates throughout their IS career.

It is then apparent that software testing should be an indispensable topic in the current IS curricula. A review of the current ACM (Nunamaker, Couger and Davis 1982) and DPMA (1986) curriculum models reveals that 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 of the ACM and CIS/86-3 and CIS/86-4 of the DPMA curriculum model. However, 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 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.

2. SOFTWARE TESTING TECHNIQUES

Conventionally, software testing techniques are classified into "black-box" and "white-box" techniques based on their methods of deriving test cases (Myers 1979, pp. 8-9). The black-box techniques derive the test cases from the requirements definition or the external (design) specification, while the white-box techniques derive them 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 focus on the structure. Therefore they are also known respectively as the functional and the structural techniques (Adrion, Branstad and Cherniavsky 1982). The test-case design methods of these two groups of techniques are briefly described below. Other techniques, such as proof of correctness, simulation, symbolic execution, and flow analysis, are excluded from our discussion because they are not directly related to test-case derivation, nor are they commonly used by the IS professionals.

2.1 Black-Box Testing 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 in which every possible value of input produces exactly the same type of output. 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 (Myers 1979, pp. 45-50).

- **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 (Myers 1979, pp. 50-55; Howden 1981). 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 the 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 (Elmendorf 1973, 1974; Myers 1979, pp. 56-57). 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 (Myers 1979, pp. 73-75). Unlike the boundary coverage technique, error guessing is largely an intuitive (Miller 1977) 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 (Adrion, Branstad and Cherniavsky 1982).

2.2 White-Box Testing Techniques

- **Statement Coverage** requires that every statement in the program be executed at least once (Miller 1977; Myers 1979, p. 38).

- **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 (Myers 1979, p. 38; Miller 1977). 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 (Myers 1979, p. 40).

- **Decision/Condition Coverage** 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 (Myers 1979, p. 41).

- **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 (Myers 1979, p. 42). Obviously, this method is superior to the above four techniques.

- **Complexity-Based Coverage**, developed by McCabe (1983), uses the "cyclomatic number" in the literature of graph theory (Harary 1969; Berge 1973; Deo 1974) 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 (McCabe 1976). 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.

3. SELECTING PRACTICAL SOFTWARE TESTING TECHNIQUES

Among the four black-box techniques, error guessing and boundary coverage are the two most commonly practiced techniques. Both techniques can help identify input conditions (valid or invalid) during equivalence partitioning, cause-effect graphing, or even walkthrough and review processes. Therefore, they should be used as supplemental techniques to all other test-case design techniques. Boundary coverage provides a structured guideline to fully test the boundary of each input condition in a program and thus is suitable for classroom training. In contrast, error guessing is not so; it does not provide any guideline for deriving test cases. Instead, it tests a program against a comprehensive checklist which was created by the tester or each individual programmer (or designer) based on his/her experience. This checklist typically focuses on the weaknesses of the individual involved (Ould and Unwin 1986) and is different from one person to another. If the one's checklist is continuously updated, it may become a very powerful and easy-to-use technique. As noted by Adrion, Branstad and Cherniavsky (1982), "guessing carries no guarantee for success, but neither does it carry any penalty." When all other techniques have failed, error guessing may come to be the only viable technique.

Between the remaining two black-box techniques, cause-effect graphing is superior to equivalence partitioning because the former 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 quickly becomes unwieldy as the problem size grows (Ould and Unwin 1986). For cause-effect graphing to be effective, it must be automated. Since currently there is no commercial tool available for cause-effect graphing today (Elmendorf 1973), we do not recommend the inclusion of cause-effect graphing in the IS curriculum. Examples of cause-effect graphing can be found in Elmendorf (1973, 1974) and Myers (1979, pp. 57-73).

As regards the white-box testing techniques, the complexity-based coverage is the best of all, because it encompasses the other five white-box techniques and further covers possible combinations of condition outcomes between any two consecutive decisions. Besides being easy to apply, complexity-based coverage can also enforce the structured programming principle that any program module (be it large or small) must have a single entry and a single exit (Mills 1972).

In summary, three out of ten existing software testing techniques are recommended for an instruction in an IS curriculum--most likely in the systems development courses. They are **equivalence partitioning, boundary coverage, and complexity-based coverage techniques.** All three techniques can be applied to a program/system of any size (be it large or small) and to manual testing such as walkthroughs (Waldstein 1974), desk checking, reviews (Freedman and Weinberg 1982), and inspections (Larson 1975; Ascoly, et al. 1976) as well as to computer-based testing such as unit tests, integration tests, system test, regression tests, conversion tests, installation tests, and acceptance tests. Since each technique has its own weaknesses, they should not be used in isolation, but rather they should supplement one another.

4. AN EXAMPLE

In order to demonstrate how to apply software testing techniques to program testing from an IS professional's perspective, an example will be walked through in detail. The example was adopted from Myers (1979, p. 1) with the exception of the last sentence, which was added to facilitate discussion. To some experienced IS professionals, the chosen example may seem trivial. Yet it allows us to have a complete and effective treatment of the recommended testing techniques and to get its basic principles across to the IS students.

Now, let's assume that an IS professional has been assigned a software 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. The process is repeated until the user decides to quit.

To accomplish the project, the IS professional should perform the following steps based on the requirements definition:

(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.

Due to the scope of this paper, we shall demonstrate only the first six steps.

Step 1: Apply the equivalence partitioning technique to the requirements definition to derive the test cases. To begin, the keywords of the requirements definition which relate to the input or output of the program 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 process is *repeated* until the user decides to *quit*.

The input keywords are *accept*, *three integers*, *keyboard*, *lengths of sides (of a triangle)*, *triangle*, *repeated*, and *quit*. The output keywords are *print*, *message*, *triangle*, *scalene*, *isosceles*, *equilateral*, *repeated*, and *quit*. Each keyword is either function-related or data-related, or both. While the keywords *accept*, *keyboard*, and *print* are strictly function-related, *three integers*, *lengths of the sides*, and *message* are data-related. The rest of the keywords, *triangle*, *scalene*, *isosceles*, *equilateral*, *repeated*, and *quit* are both function-related and data-related. Our focus is on the five keywords which are either data-related or both: *three integers*, *lengths of sides*, *triangle*, *repeated*, and *quit*. The first keyword indicates that there are three integers to be processed; the second indicates that these three integers are the lengths of sides of a triangle and thus must be great than zero; the third implies that the sum of the lengths of any two sides of a triangle is greater than that of the third side. The fourth and the fifth keywords, *repeated* and *quit*, indicate a loop condition exists which has two possible outcomes, to repeat or to quit, depending on the input condition. Based on these keywords, the possible valid and invalid input conditions and their corresponding expected output conditions are enumerated in Table 1. Each combination of input 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 in Parentheses)*
Valid:	
Three Integers:	
1. A is an integer	1. a triangle (1-9,11)*
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	
Loop:	
10. repeat	2. repeat the process (10)
11. quit	
Invalid:	
12. A<1 & A is an integer	3. invalid integer A (12,11)*
13. B<1 & B is an integer	4. invalid integer B (13,11)*
14. C<1 & C is an integer	5. invalid integer C (14,11)*
15. A is not an integer	6. non-integer A (15,11)*
16. B is not an integer	7. non-integer B (16,11)*
17. C is not an integer	8. non-integer C (17,11)*
18. A+B≤C	9. not a triangle (18,11)*
19. A+C≤B	10. not a triangle (19,11)*
20. B+C≤A	11. not a triangle (20,11)*

* This test is executed without any repetition.

Step 2 One possible set of pseudocode for this program is listed below. Note that pseudocoding in the program's internal specification emphasizes not the *efficiency* (structure) but the *effectiveness* (functions) of the desired program. The pseudocode presented here may not be efficient, but it is effective enough to perform the intended functions.

```
PROGRAM TRIANGLE(A,B,C):
Loop: 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."
    ACCEPT the value of the repetition flag (R).
    REPEAT Loop UNTIL R=FALSE.
    END of program.
```

Step 3: The above pseudocode can be represented by a control-flow graph. A control-flow graph is a directed graph having each of its vertices represent a code segment in the program (i.e., a sequence of consecutive statements with a single entry and a single exit) and each edge represent a possible transfer of control from one segment to another. Figure 1 shows the control-flow graph representing the program pseudocode. The graph is drawn with McCabe's (1983) convention which uses multiple branches to represent the true/false outcomes of a compound decision (i.e., a decision with AND or OR operators). To facilitate identifying test paths, each decision branch in Figure 1 is labeled with an alphabet starting at "*a*" and each decision node with an integer starting at "*1*."

Step 4: This step is to develop a set of test cases using the complexity-based coverage technique introduced by McCabe (1983) 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. The procedure of complexity-based coverage as it applies to the control-flow graph in Figure 1 is described below:

(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 path that performs the major full function provided in the program and intersects a maximal number of decisions in the graph, as opposed to an error path that results in an error message or recovery procedure. For example, path *1d2h3i4k6p7r* is a possible baseline. Note that our path expression is somewhat different than that of McCabe (1983) in which the decision number does not appear.

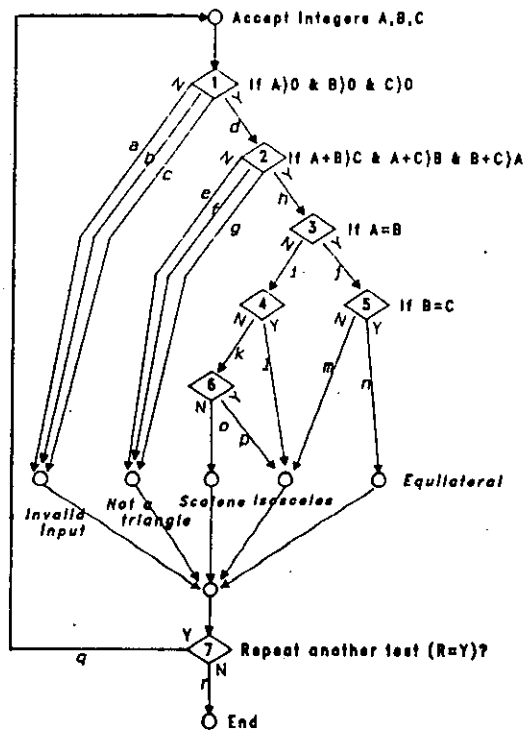

Figure 1. Control-Flow Graph for the Triangle Program.

(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: *~1a7r*, *~1b7r*, and *~1c7r*. 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~2e7r*, *1d~2f7r*, and *1d~2g7r*.

(4) Repeat the above 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~3j5m7r*. Flipping the fourth decision yields the path *1d2h3i~4l7r*; the sixth decision yields the path *1d2h3i4k~6o7r*; the seventh decision yields *1d2h3i4k6p(~7q1d2f7r)*. The parenthesized segment on the last path represents the boundary and the in-

terior decisions of the loop. Since McCabe did not provide any guideline for selecting the path inside the loop, we have decided to take the functional path *~7q1d2f* which is equivalent to the test case *10* in Table 1.

(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) before we stop. Flipping the fifth decision yields the path *1d2h~3j~5n7r.*

Table 2 shows the 12 paths derived by the complexity-based coverage technique and their corresponding test-case numbers from Table 1. Notice that the number of test cases derived from this procedure (which is 12) always equals the cyclomatic number of the program which is one plus the number of decision conditions in the program (which is 11).

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.	1d2h3i4k6p7r (Baseline)	1*
2.	~1a7r	3
3.	~1b7r	4
4.	~1c7r	5
5.	1d~2e7r	9
6.	1d~2f7r	10
7.	1d~2g7r	11
8.	1d2h~3j5m7r	1*
9.	1d2h3i~417r	1*
10.	1d2h3i4k~6o7r	1*
11.	1d2h3i4k6p(~7q1d2f7r)	1**,2,10
12.	1d2h~3j~5n7r	1*
13.	***	6***
14.	***	7***
15.	***	8***

* Unlike the equivalence partitioning technique, the pseudocode as well as the control-flow graph considered the input conditions for different types (i.e., outcomes) of triangle. Therefore, this test case corresponds to many complexity-based test paths.

** Case 1 does not cover the "repeat" action here but rather it covers the "quit" one.

*** 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 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 triangles as the complexity-based coverage did. Since our objective is to derive and use test cases as complete as possible, we shall consolidate the two set of test cases and use all the 15 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 such as 1) single entry and exit, 2) no unconditional GOTO branch, 3) the use of structured constructs, 4) modularization, etc. (Bohm and Jacopini 1966; Dijkstra 1968; 1970; Mills 1972). Moreover, the cyclomatic number of a program should have an upper bound of 10, as suggested by McCabe (1976). Otherwise, the number of possible test paths could become unmanageable.

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 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 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.

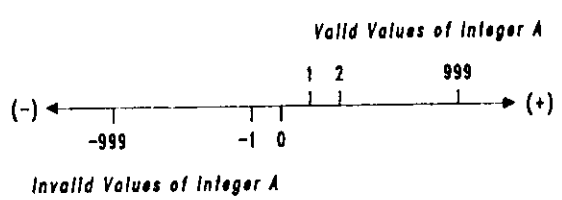

Figure 2. The Boundary Values of the Input Integer A

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	Case I.D. in Table 1	Expected Test Outcomes	Test Data Derived by Boundary Coverage for Each Test Case*
1	~1a7r	3	Invalid A	A= 0 B= ** C= ** R=N
2				A= -1 B= ** C= ** R=N
3	~1b7r	4	Invalid B	A= 1 B= 0 C= ** R=N
4				A= 1 B= -1 C= ** R=N
5	~1c7r	5	Invalid C	A= 1 B= 1 C= 0 R=N
6				A= 1 B= 1 C= -1 R=N
7	***	6	Non-integer A	A=1.01 B= ** C= ** R=N
8				A=.999 B= ** C= ** R=N
9	***	7	Non-integer B	A= 1 B=1.01 C= ** R=N
10				A= 1 B=.999 C= ** R=N
11	***	8	Non-integer C	A= 1 B= 1 C=1.01 R=N
12				A= 1 B= 1 C=.999 R=N
13	1d~2e7r	9	Non-triangle	A= 1 B= 1 C= 2 R=N
14				A= 1 B= 1 C=999 R=N
15	1d~2f7r	10	Non-triangle	A= 1 B= 2 C= 1 R=N
16				A= 1 B=999 C= 1 R=N
17	1d~2g7r	11	Non-triangle	A= 2 B= 1 C= 1 R=N
18				A=999 B= 1 C= 1 R=N
19	1d2h3i4k6p7r	1	Isosceles	A= 2 B= 1 C= 2 R=N
20	1d2h~3j5m7r	1	Isosceles	A=999 B=999 C= 1 R=N
21	1d2h3i~417r	1	Isosceles	A= 1 B=999 C=999 R=N
22	1d2h3i4k~6o7r	1	Scalene	A= 2 B= 3 C= 4 R=N
23	1d2h3i4k6p(~7q1d2f7r)	1,2,10	Isosceles & repeat, then non-triangle	A=999 B= 1 C=999 R=Y A= 2 B= 4 C= 2 R=N
24	1d2h~3j~5n7r	1	Equilateral	A= 1 B= 1 C= 1 R=N

* Without boundary-value analysis, the data may not be the same and the second set of test data for each invalid input condition may not be generated.

** This entry can be of any value.

*** No corresponding test path is generated because the integer format is assumed to be checked by the system.

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 the expected output space in our example is not completely covered by the input equivalence classes. Not every expected unique type of triangle (see column 4 of Table 3) has a matching input equivalence class (see column 3 of Table 3). However, this problem was overcome by consolidating the test cases derived from equivalence partitioning with those from the complexity-based coverage (see column 2 of Table 3). 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 the 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 (Kernighan and Plauger 1974).

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, displays a menu of module numbers, names, and functional descriptions, and prompts for the user's selection of one of the module numbers, 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 (skip the loop), exactly 1 entry (no iteration), and 2 or more entries (some iterations). This coverage method is known as the "boundary-interior" path testing procedure (Howden 1975).

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 24 test-case items listed in Table 3 must be executed. If any major error was found during the testing process, the error should be removed before the process is repeated. This testing process should be terminated only when all test results are identical to the expected results. Note that the test paths derived by the complexity-based coverage varies as the programming style or logic flow changes. In contrast, equivalence partitioning does not require the test-case designer to know any program code and thus is independent of programming style and logic flow.

5. A PEDAGOGICAL EXPERIENCE

The process demonstrated above is highly structured and straightforward. Therefore, it is pedagogically feasible for classroom instruction and practices. We have imparted this process to our students in the system design and implementation course and received overwhelmingly positive feedback. Our experience indicates that, before learning the three recommended testing techniques, most students who did not have training in software testing were exclusively using the error guessing technique--which is more a form of cynicism than a technique--to derive test cases and data. After practicing the above testing process for two or three exercises, all of them eventually became effective test-case designers.

6. CONCLUSION AND DISCUSSION

Software quality is one of the major factors influencing the quality of information systems 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 participates in a software project must be ready to participate in not only 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) but also 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, an IS graduate is required to have knowledge of software testing techniques.

This paper reviews the existing software testing techniques and recommends a set of effective techniques which are essential to the IS professionals in testing their IS software. The techniques recommended include *equivalence partitioning*, *boundary coverage*, and *complexity-based coverage*. These three testing techniques provide structured approaches to designing test cases and data for testing the quality of a software product. Therefore, they are of vital importance to every practicing IS professional, similar to such structured techniques as structured analysis, structured design, and structured programming.

It is important to note that software testing is by no means the only process that can assure the quality of a software product. Assuring software quality is a collective effort of all the processes in an SQA program, i.e., configuration management, corrective action, documentation, reviews, and audits must be performed besides software testing. In fact, the ultimate level of quality is not determined by the testing process but by the development process itself. The probability of success (quality) at acceptance time is a function of the tools, standards, practices, and procedures used by the development organization augmented by the SQA processes at built-in quality check points in the development process (Knight 1977). All the required tools and methods must be defined at the outset of the software project to allow software quality be objectively measured at the planned check points. Since most software errors (60 to 80 percent) were found to be associated with the requirements definition (Boar 1984), some of these check points should be placed at the early phases of the software project. To quote an old sage, "Quality is built in, not added on." Software testing should start as soon as the software project begins and software quality should be closely designed, measured, and maintained throughout the entire project life cycle.

7. REFERENCES

Adrion, W. R.; Branstad, M. A.; and Cherniavsky, J. C. "Validation, Verification, and Testing of Computer Software." *ACM Computing Surveys*, Vol. 14, No. 2, June 1982, pp. 159-192.

Ardis, M. A. "The Evolution of Wang Institute's Master of Software Engineering Program." *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 11, November 1987, pp. 1149-1155.

Ascoly, J.; Cafferty, M.; Gruen, S.; and Kohli, O. "Code Inspection Specification." TR-21.630, IBM System Communications Division, Kingston, New York, 1976.

Berge, C. *Graphs and Hypergraphs*. Amsterdam: North-Holland, 1973, pp. 15-17.

Boar, B. H. *Application Prototyping: A Requirements Definition Strategy for the 80s*. New York, NY: Wiley-Interscience, 1984, pp. 17-18.

Bohm, C., and Jacopini, G. "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules." *Communications of the ACM*, Vol. 9, No. 5, May 1966, pp. 366-371.

Deo, N. *Graph Theory with Applications to Engineering and Computer Science*. Englewood Cliffs, NJ: Prentice-Hall, 1974, pp. 55-58.

Dijkstra, E. W. "Go To Statement Considered Harmful." *Communications of the ACM*, Vol. 11, No. 3, March 1968, pp. 147-148.

DPMA. *The DPMA Model Curriculum for Undergraduate Computer Information Systems*, Second Edition. Park Ridge, IL: Data Processing Management Association, July 1986.

Elmendorf, W. R. "Cause-Effect Graphs in Functional Testing." TR-00.2487, IBM System Development Division, Poughkeepsie, New York, 1973.

Elmendorf, W. R. "Functional Analysis Using Cause-Effect Graphs." *Proceedings of SHARE XLIII*, New York, 1974, pp. 567-577.

Freedman, D. P., and Weinberg, G. M. *Handbook of Walkthroughs, Inspections, and Technical Reviews*. Boston, MA: Little, Brown and Company, 1982, pp. 19-30.

Freeman, P. "Essential Elements of Software Engineering Education Revisited." *IEEE Transactions on Software Engineering*, Vol. SE-13, No. 11, November 1987, pp. 1143-1148.

Gustafson, G. G., and Kerr, R. J. "Some Practical Experience with A Software Quality Assurance Program." *Communications of the ACM*, Vol. 25, No. 1, January 1982, pp. 4-12.

Harary, F. *Graph Theory*. Reading, MA: Addison-Wesley, 1969, pp. 37-40.

Howden, W. E. "Methodology for the Generation of Program Test Data." *IEEE Transactions on Computers*, Vol. C-24, No. 5, May 1975, pp. 554-559.

Howden, W. E. "A Survey of Dynamic Analysis Methods." In E. Miller and W. E. Howden (eds.), *Tutorial: Software Testing and Validation Techniques*. New York: IEEE Computer Society, 1981, pp. 209-231.

IEEE Computer Society. "Curriculum Recommendations for Software Engineering." IEEE Computer Society, Los Alamitos, California, 1982.

Kernighan, B. W., and Plauger, P. J. *The Elements of Programming Style*. New York, NY: McGraw-Hill, 1974, pp. 61-62 & 89.

Knight, B. M. "Software Quality Assurance Implementation of A MIL-S-52779 Program." *Proceedings of NSIA Quality and Reliability Assurance Committee Conference on Software Quality-Reliability*, Arlington, Virginia, March 1977, pp. 7.1-7.3.

Larson, R. R. "Test Plan and Test Case Inspection Specifications." TR-21.586, IBM System Development Division, Kingston, New York, 1975.

McCabe, T. J. "A Complexity Measure." *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, April 1976, pp. 308-320.

McCabe, T. J. "A Testing Methodology Using the McCabe Complexity Metric." In T.J. McCabe (ed.), *Structured Testing*, IEEE Computer Society Press, Silver Spring, Maryland, 1983, pp. 19-47.

Miller, E. F., Jr. "Program Testing: Art Meets Theory." *Computer*, Vol. 10, No. 7, July 1977, pp. 42-51.

Mills, H. D. "Mathematical Foundations for Structured Programming." FSC 72-6012, IBM Federal System Division, Gaithersburg, Maryland, 1972.

Myers, G. J. The Art of Software Testing. New York, NY: Wiley-Interscience, 1979.

Nunamaker, J. F. (ed.). "Educational Programs in Information Systems." *Communications of the ACM*, Vol. 24, No. 3, March 1981, pp. 124-133.

Nunamaker, J. F.; Couger, J. D.; and Davis, G. B. (eds.). "Information Systems Curriculum Recommendations for the 80s: Undergraduate and Graduate Programs." *Communications of the ACM*, Vol. 25, No. 11, November 1982, pp. 781-805.

Ould, M. A., and Unwin, C. *Testing in Software Development*. Cambridge, England: Cambridge University Press, 1986, pp. 80-99.

Pfau, P. R. "Applied Quality Assurance Methodology." *Proceedings of the Software Quality Assurance Workshop*, San Diego, California, November 1978, 1-8.

Stamm, S. L. "Assuring Quality Quality Assurance." *Datamation*, March 1981, pp. 195-200.

Waldstein, N. S. "The Walk-Thru: A Method of Specification, Design and Code Review." TR-00-2536, IBM System Development Division, Poughkeepsie, New York, 1974.