ITE 368: Software Testing and Maintenance
Week 2: Test-Case Design (1)

Ehsan Ali
Stamford International University (Thailand)
ehsan.ali@stamford.edu

Semester 3/2021

Contents

Chapter 1

Week 2: Test-Case Design

1.1 Introduction

Moving beyond the psychological issues discussed in Week 1, the most important consideration in program testing is the design and creation of effective **test cases**. Test-case design is so important because complete testing is impossible.

Given constraints on time and cost, the key issue of testing becomes: What subset of all possible test cases has the highest probability of detecting the most errors?

In general, the least effective methodology of all is *random-input testing*. a randomly selected collection of test cases has little chance of being an optimal, or even close to optimal, subset. Therefore, in this week, we want to develop a set of thought processes that enable you to select test data more intelligently.

We saw that exhaustive black-box and white-box testing are, in general, impossible; at the same time, it suggested that a reasonable testing strategy might feature elements of both. This is the strategy that we develop this week.

You can develop a reasonably rigorous test by using certain *black-box–oriented* test-case design methodologies and then supplementing these test cases by examining the logic of the program, using *white-box* methods.

The methodologies discussed in this week are:

Black Box	**White Box**
Equivalence partitioning	Statement coverage
Boundary value analysis	Decision coverage
Cause-effect graphing	Condition coverage
Error guessing	Decision/condition coverage
	Multiple-condition coverage

Table 1.1: Black-box and white-box methodologies.

It is recommended that you use a combination of most, if not all, of them to design a rigorous test of a program, since each method has distinct strengths and weaknesses.

1.2 White-Box Testing

The ultimate white-box test is the execution of every path in the program; but complete path testing is not a realistic goal for a program with loops.

1.2.1 Logic Coverage Testing

If you back completely away from path testing, it may seem that a worthy goal would be to execute every statement in the program at least once. Unfortunately, this is a weak criterion for a reasonable white-box test. This concept is illustrated in Fig. 1.1. Assume that this figure represents a small program to be tested. The equivalent Java code snippet follows:

```java
public void foo(int A, int B, int X) {
  if(A > 1 && B == 0) {
    X = X / A;
  }
  if(A == 2 && X > 1) {
    X = X + 1;
  }
}
```

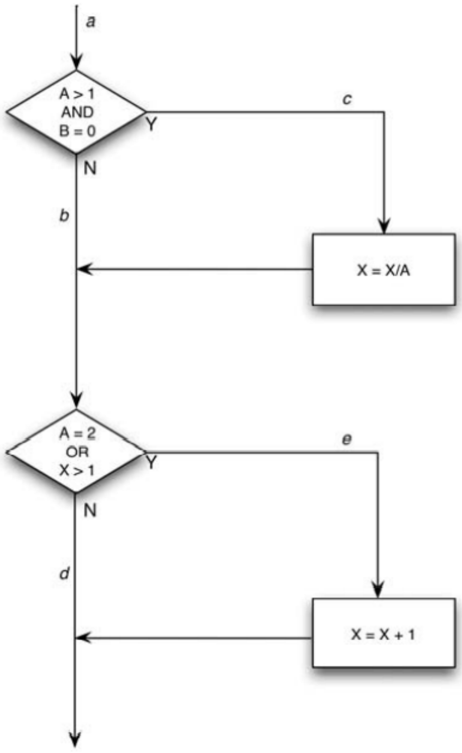


Figure 1.1: A Small Program to Be Tested.

You could execute every statement by writing a single test case that traverses path `ace`. That is, by setting A=2, B=0, and X=3 at point `a`, every statement would be executed once (actually, `X` could be assigned any integer value > 1).

Unfortunately, this criterion is a rather poor one. For instance, perhaps the first decision should be an `or` rather than an `and`. If so, this error would go undetected. Perhaps the second decision should have stated X>0; this error would not be detected. Also, there is a path through the program in which `X` goes unchanged (the path `abd`). If this were an error, it would go undetected. In other words, the statement coverage criterion is so weak that it generally is useless.

A stronger logic coverage criterion is known as **decision coverage** or **branch coverage**. This criterion states that you must write enough test cases that each decision has a true and a false outcome at least once. In other words, each branch direction must be traversed at least once. Examples of branch or decision statements are *switch-case*, *do-while*, and *if-else* statements. Multipath `GOTO` statements qualify in some programming languages such as Fortran.

Decision coverage usually can satisfy statement coverage. Since every statement is on some subpath flowing either from a branch statement or from the entry point of the program, every statement must be executed if every branch direction is executed. There are, however, at least three exceptions:

- Programs with no decisions.
- Programs or subroutines/methods with multiple entry points. A given statement might be executed only if the program is entered at a particular entry point.
- Statements within ON-units. Traversing every branch direction will not necessarily cause all ON-units to be executed.

Note: An ON-unit is a method of exception handling in the PL/I language `https://en.wikipedia.org/wiki/PL/I`, like a `catch`block in more modern languages.

Since we have deemed statement coverage to be a necessary condition, decision coverage, a seemingly better criterion, should be defined to include statement coverage. Hence, decision coverage requires that each decision have a true and a false outcome, and that each statement be executed at least once. An alternative and easier way of expressing it is that each decision has a true and a false outcome, and that each point of entry (including ON-units) be invoked at least once.

This discussion considers only two-way decisions or branches and has to be modified for programs that contain *multipath decisions*. Examples are Java programs containing *switch-case* statements, Fortran programs containing arithmetic (three-way) IF statements or computed or arithmetic `GOTO` statements, and COBOL programs containing altered `GOTO` statements or `GO-TO-DEPENDING-ON` statements. For such programs, the criterion is exercising each possible outcome of all decisions at least once and invoking each point of entry to the program or subroutine at least once.

In Fig. 1.1, decision coverage can be met by two test cases covering paths `ace` and `abd` or, alternatively, `acd` and `abe`. If we choose the latter alternative, the two test-case inputs are A=3, B=0, X=3 and A=2, B=1, and X=1.

Decision coverage is a stronger criterion than statement coverage, but it still is rather weak. For instance, there is only a 50 percent chance that we would explore the path where `X` is not changed (i.e., only if we chose the former alternative). If the second decision were in error (if it should have said X<1 instead of X>1), the mistake would not be detected by the two test cases in the previous example.

A criterion that is sometimes stronger than decision coverage is **condition coverage** . In this case, you write enough test cases to ensure that each condition in a decision takes on all possible outcomes at least once. But, as with decision coverage, this does not always lead to the execution of each statement, so an addition to the criterion is that each point of entry to the program or subroutine, as well as ON-units, be invoked at least once. For instance, the branching statement:

```
DO K=0 to 50 WHILE (J+K < QUEST)
```

contains two conditions: Is `K` less than or equal to 50, and is `J+K` less than `QUEST`? Hence, test cases would be required for the situations `K<=50`, `K>50` (to reach the last iteration of the loop), `J+K<QUEST`, and `J+K>=QUEST`.

Fig. 1.1 has four conditions:`A>1`, `B=0`, `A=2`, and `X>1`. Hence, enough test cases are needed to force the situations where `A>1`, `A<=1`, `B=0`, and `B<>0` are present at point `a` and where `A=2`, `A<>2`, `X>1`, and X<=1 are present at point b. A sufficient number of test cases satisfying the criterion, and the paths traversed by each, are:

```
A=2, B=0, X=4 ace
A=1, B=1, X=1 adb
```

Note that although the same number of test cases was generated for this example, condition coverage usually is superior to decision coverage in that it may (but does not always) cause every individual condition in a decision to be executed with both outcomes, whereas decision coverage does not.

For instance, in the same branching statement:

```
DO K=0 to 50 WHILE (J+K < QUEST)
```

is a two-way branch (execute the loop body or skip it). If you are using *decision testing*, the criterion can be satisfied by letting the loop run from K=0 to 51, without ever exploring the circumstance where the WHILE clause becomes false. With the *condition criterion*, however, a test case would be needed to generate a false outcome for the conditions J+K < QUEST.

Although the condition coverage criterion appears, at first glance, to satisfy the decision coverage criterion, it does not always do so. If the decision IF(A & B) is being tested, the condition coverage criterion would let you write two test cases—A is true, B is false, and A is false, B is true—but this would not cause the THEN clause of the IF to execute. The condition coverage tests for the earlier example covered all decision outcomes, but this was only by chance. For instance, two alternative test cases

```
A=1, B=0, X=3
A=2, B=1, X=1
```

cover all condition outcomes but only two of the four decision outcomes (both of them cover path abe and, hence, do not exercise the true outcome of the first decision and the false outcome of the second decision).

The obvious way out of this dilemma is a criterion called **decision/condition coverage**. It requires sufficient test cases such that each condition in a decision takes on all possible outcomes at least once, each decision takes on all possible outcomes at least once, and each point of entry is invoked at least once.

A weakness with decision/condition coverage is that although it may appear to exercise all outcomes of all conditions, it frequently does not, because certain conditions mask other conditions. To see this, examine Fig 1.2. The flowchart in this figure is the way a compiler would generate machine code for the program in Fig. 1.1. The multicondition decisions in the source program have been broken into individual decisions and branches because most machines do not have a single instruction that makes multicondition decisions. A more thorough test coverage, then, appears to be the exercising of all possible outcomes of each primitive decision. The two previous decision coverage test cases do not accomplish this; they fail to exercise the false outcome of decision H and the true outcome of decision K.

The reason, as shown in Fig. 1.2, is that results of conditions in the and and the or expressions can mask or block the evaluation of other conditions. For instance, if an and condition is false, none of the subsequent conditions in the expression need be evaluated. Likewise, if an or condition is true, none of the subsequent conditions need be evaluated. Hence, errors in logical expressions are not necessarily revealed by the condition coverage and decision/condition coverage criteria.

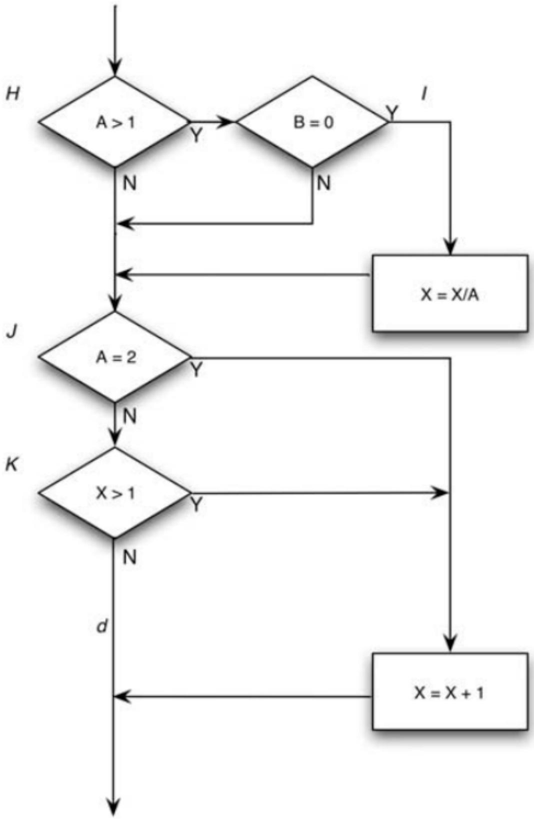


Figure 1.2: Machine Code for the Program in Fig. 1.1

A criterion that covers this problem, is **multiple-condition coverage**. This criterion requires that you write sufficient test cases such that all possible combinations of condition outcomes in each decision, and all points of entry, are invoked at least once. For instance, consider the following sequence of pseudo-code.

```
NOTFOUND = TRUE;
DO I=1 to TABSIZE WHILE (NOTFOUND); /*SEARCH TABLE*/
.. . searching logic.. . ;
END
```

The four situations to be tested are:

1. I<=TABSIZE and NOTFOUND is true.
2. I<=TABSIZE and NOTFOUND is false (finding the entry before hitting the end of the table).
3. I>TABSIZE and NOTFOUND is true (hitting the end of the table without finding the entry).
4. I>TABSIZE and NOTFOUND is false (the entry is the last one in the table).

It should be easy to see that a set of test cases satisfying the multiple-condition criterion also satisfies the decision coverage, condition coverage, and decision/condition coverage criteria.

Returning to Fig. 1.1, test cases must cover eight combinations:

1. A>1, B=0

2. `A>1, B<>0`
3. `A<=1, B=0`
4. `A<=1, B<>0`
5. `A=2, X>1`
6. `A=2, X<=1`
7. `A<>2, X>1`
8. `A<>2, X<=1`

Note Recall from the Java code snippet presented earlier that test cases 5 through 8 express values at the point of the second `if` statement. Since `X` may be altered above this `if` statement, the values needed at this `if` statement must be backed up through the logic to find the corresponding input values.

These combinations to be tested do not necessarily imply that eight test cases are needed. In fact, they can be covered by four test cases. The test-case input values, and the combinations they cover, are as follows:

1. `A=2, B=0, X=4` Covers 1, 5
2. `A=2, B=1, X=1` Covers 2, 6
3. `A=1, B=0, X=2` Covers 3, 7
4. `A=1, B=1, X=1` Covers 4, 8

The fact that there are four test cases and four distinct paths in Fig. 1.1 is just coincidence. In fact, these four test cases do not cover every path; they miss the path `acd`. For instance, you would need eight test cases for the following decision:

```
if(x==y && length(z)==0 && FLAG) {
  j=1;
else
  i=1;
}
```

although it contains only two paths. In the case of loops, the number of test cases required by the multiple-condition criterion is normally much less than the number of paths.

In summary, for programs containing only one condition per decision, a minimum test criterion is a sufficient number of test cases to:

1. invoke all outcomes of each decision at least once
2. invoke each point of entry (such as entry point or ON-unit) at least once

to ensure that all statements are executed at least once.

For programs containing decisions having multiple conditions, the minimum criterion is a sufficient number of test cases to invoke all possible combinations of condition outcomes in each decision, and all points of entry to the program, at least once. (The word "possible" is inserted because some combinations may be found to be impossible to create.)

1.3 Black-Box Testing

The Black-box (data-driven or input/output driven) testing is based on program specifications. The goal is to find areas wherein the program does not behave according to its specifications.

1.3.1 Equivalence Partitioning

We learned that an exhaustive input test of a program is impossible. Hence, when testing a program, you are limited to a small subset of all possible inputs. Of course, then, you want to select the ”right” subset, that is, the subset with the highest probability of finding the most errors.

One way of locating this subset is to realize that a well-selected test case also should have two other properties:

1. It reduces, by more than a count of one, the number of other test cases that must be developed to achieve some predefined goal of “reasonable” testing.

2. It covers a large set of other possible test cases. That is, it tells us something about the presence or absence of errors over and above this specific set of input values.

These properties, although they appear to be similar, describe two distinct considerations. The first implies that each test case should invoke as many different input considerations as possible to minimize the total number of test cases necessary. The second implies that you should try to partition the input domain of a program into a finite number of equivalence classes such that you can reasonably assume (but, of course, not be absolutely sure) that a test of a representative value of each class is equivalent to a test of any other value. That is, if one test case in an equivalence class detects an error, all other test cases in the equivalence class would be expected to find the same error. Conversely, if a test case did not detect an error, we would expect that no other test cases in the equivalence class would fall within another equivalence class, since equivalence classes may overlap one another.

These two considerations form a black-box methodology known as **equivalence partitioning**. The second consideration is used to develop a set of “interesting” conditions to be tested. The first consideration is then used to develop a minimal set of test cases covering these conditions.

An example of an equivalence class in the triangle program of Week 1 is the set “three equal-valued numbers having integer values greater than zero.” By identifying this as an equivalence class, we are stating that if no error is found by a test of one element of the set, it is unlikely that an error would be found by a test of another element of the set. In other words, our testing time is best spent elsewhere: in different equivalence classes.

Test-case design by equivalence partitioning proceeds in two steps:

1. identifying the equivalence classes

2. defining the test cases.

1.3.2 Identifying the Equivalence Classes

The *equivalence classes* are identified by taking each input condition (usually a sentence or phrase in the specification) and partitioning it into two or more groups. You can use the table in Fig. 1.3 to do this. Notice that two types of equivalence classes are identified: *valid equivalence classes* represent valid inputs to the program, and *invalid equivalence classes* represent all other possible states of the condition (i.e., erroneous input values).

Thus, we are adhering to principle 5, discussed in Week 1, which stated you must focus attention on invalid or unexpected conditions.

Given an input or external condition, identifying the equivalence classes is largely a heuristic process. Follow these guidelines:

1. If an input condition specifies a range of values (e.g., “the item count can be from 1 to 999”), identify one valid equivalence class (1<item count<999) and two invalid equivalence classes (item count<1 and item count>999).

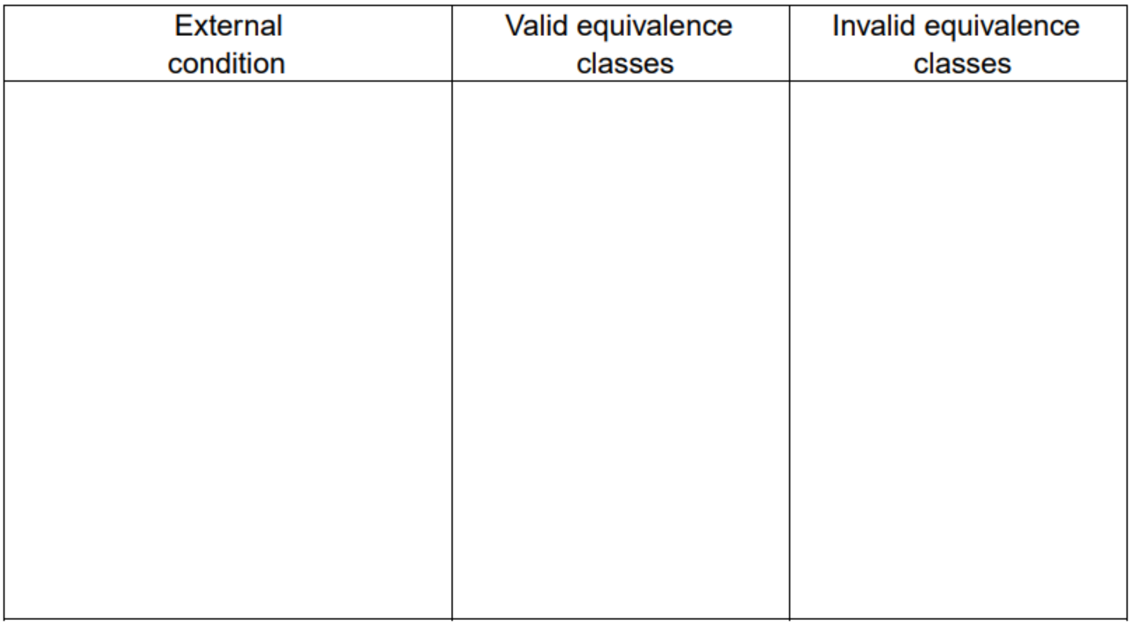
2. If an input condition specifies the number of values (e.g., “one through six owners can be listed for the automobile”), identify one valid equivalence class and two invalid equivalence classes (no owners and more than six owners).

3. If an input condition specifies a set of input values, and there is reason to believe that the program handles each differently (“type of vehicle must be BUS, TRUCK, TAXICAB, PASSENGER, or MOTORCYCLE”), identify a valid equivalence class for each and one invalid equivalence class (“TRAILER,” for example).

4. If an input condition specifies a “must-be” situation, such as “first character of the identifier must be a letter,” identify one valid equivalence class (it is a letter) and one invalid equivalence class (it is not a letter).

If there is any reason to believe that the program does not handle elements in an equivalence class identically, split the equivalence class into smaller equivalence classes. We will illustrate an example of this process shortly.

Note: A heuristic technique, is any approach to problem solving or self-discovery that employs a practical method that is not guaranteed to be optimal, perfect, or rational, but is nevertheless sufficient for reaching an immediate, short-term goal or approximation.



External condition	Valid equivalence classes	Invalid equivalence classes

Figure 1.3: A Form for Enumerating Equivalence Classes. 1.1

1.3.3 Identifying the Test Cases

The second step is the use of equivalence classes to identify the test cases. The process is as follows:

1. Assign a unique number to each equivalence class.

2. Until all valid equivalence classes have been covered by (incorporated into) test cases, write a new test case covering as many of them uncovered valid equivalence classes as possible.

3. Until your test cases have covered all invalid equivalence classes, write a test case that covers one, and only one, of the uncovered invalid equivalence classes.

The reason that *individual test cases* cover invalid cases is that certain erroneous-input checks mask or supersede other erroneous-input checks. For instance, if the specification states "enter book type (HARDCOVER, SOFTCOVER, or LOOSE) and amount (1–999)," the test case, (XYZ 0), expressing two error conditions (invalid book type and amount) will probably not exercise the check for the amount, since the program may say "XYZ IS UNKNOWN BOOK TYPE" and not bother to examine the remainder of the input.

1.3.4 Equivalence Partitioning - Example

As an example, assume that we are developing a compiler for a subset of the Fortran language, and we wish to test the syntax checking of the `DIMENSION` statement. The specification is listed below. (Note: This is not the full Fortran DIMENSION statement; it has been edited considerably to make it simpler. Do not be deluded into thinking that the testing of actual programs is as easy as the examples provided here.)

In the specification, items in italics indicate syntactic units for which specific entities must be substituted in actual statements; brackets are used to indicate option items; and an ellipsis (...) indicates that the preceding item may appear multiple times in succession.

Specification:

```
A DIMENSION statement is used to specify the dimensions of arrays.
The form of the DIMENSION statement is

DIMENSION ad [,ad] ...

   where ad is an array descriptor of the form n(d[,d]...) where
n is the symbolic name of the array and d is a dimension declarator.
Symbolic names can be one to six letters or digits, the first of
which must be a letter.  The minimum and maximum numbers of dimension
declarations that can be specified for an array are one and seven,
respectively.  The form of a dimension declarator is

[lb: ]ub

   where lb and ub are the lower and upper dimension bounds.  A
bound may be a constant in the range -65534 to 65535 or the name
of an integer variable (but not an array element name).  If lb
is not specified, it is assumed to be 1.  The value of ub must
be greater than or equal to lb.  If lb is specified, its value
may be negative, 0, or positive.  As for all statements, the DIMENSION
statement may be continued over multiple lines.
```

The first step is to identify the input conditions and, from these, locate the equivalence classes. These are tabulated in Table 1.2. The numbers in the table are unique identifiers of the equivalence classes. The next step is to write a test case covering one or more valid equivalence classes. For instance, the test case

```
DIMENSION A(2)
```

covers classes 1, 4, 7, 10, 12, 15, 24, 28, 29, and 43.

Table 1.2: Equivalence Classes

Input Condition	**Valid Equivalence Classes**	**Invalid Equivalence Classes**
Number of array descriptors	one (1), > one (2)	none (3)
Size of array name	1–6 (4)	0 (5), >6 (6)
Array name	has letters (7), has digits (8)	has something else (9)
Array name starts with letter	yes (10)	no (11)
Number of dimensions	1–7 (12)	0 (13), >7 (14)
Upper bound is	constant (15), integer variable (16)	array element name (17), something else (18)
Integer variable name	has letter (19), has digits (20)	has something else (21)
Integer variable starts with letter	yes (22)	no (23)
Constant	–65534 –65535 (24)	< –65534 (25), > 65535 (26)
Lower bound specified	yes (27), no (28)	
Upper bound to lower bound	greater than (29), equal (30)	less than (31)
Specified lower bound	negative (32), zero (33), > 0 (34)	
Lower bound is	constant (35), integer variable (36)	array element name (37), something else (38)
Lower bound is	one (39)	ub> 1 (40), ub < 1 (41)
Multiple lines	yes (42), no (43)	

The next step is to devise one or more test cases covering the remaining valid equivalence classes. One test case of the form

```
DIMENSION A12345 (I,9,J4XXXX,65535,1,KLM,
  X), BBB(-65534:100,0:1000,10:10, I:65535)
```

covers the remaining classes. The invalid input equivalence classes, and a test case representing each, are:

```
(3): DIMENSION
(5): DIMENSION (10)
(6): DIMENSION A234567(2)
(9): DIMENSION A.1(2)
(11): DIMENSION 1A(10)
(13): DIMENSION B
(14): DIMENSION B(4,4,4,4,4,4,4,4)
(17): DIMENSION B(4,A(2))
(18): DIMENSION B(4,,7)
(21): DIMENSION C(I.,10)
(23): DIMENSION C(10,1J)
(25): DIMENSION D(-65535:1)
```

```
(26): DIMENSION D(65536)
(31): DIMENSION D(4:3)
(37): DIMENSION D(A(2):4)
(38): D(.:4)
(43): DIMENSION D(0)
```

Hence, the equivalence classes have been covered by 17 test cases.

Although equivalence partitioning is vastly superior to a random selection of test cases, it still has deficiencies. It overlooks certain types of high-yield test cases, for example. The next two methodologies, boundary value analysis and cause-effect graphing, cover many of these deficiencies.

1.3.5 Boundary Value Analysis

Experience shows that test cases that explore **boundary conditions** have a higher payoff than test cases that do not. Boundary conditions are those situations directly on, above, and beneath the edges of input equivalence classes and output equivalence classes. Boundary value analysis differs from equivalence partitioning in two respects:

1. Rather than selecting any element in an equivalence class as being representative, boundary value analysis requires that one or more elements be selected such that each edge of the equivalence class is the subject of a test.

2. Rather than just focusing attention on the input conditions (input space), test cases are also derived by considering the result space (output equivalence classes).

It is difficult to present a ”cookbook” for boundary value analysis, since it requires a degree of creativity and a certain amount of specialization toward the problem at hand. (Hence, like many other aspects of testing, it is more a state of mind than anything else.) However, a few general guidelines are in order:

1. If an input condition specifies a range of values, write test cases for the ends of the range, and invalid-input test cases for situations just beyond the ends. For instance, if the valid domain of an input value is –1.0 to 1.0, write test cases for the situations –1.0, 1.0, –1.001, and 1.001.

2. If an input condition specifies a number of values, write test cases for the minimum and maximum number of values and one beneath and beyond these values. For instance, if an input file can contain 1–255 records, write test cases for 0, 1, 255, and 256 records.

3. Use guideline 1 for each output condition. For instance, if a payroll program computes the monthly social security deduction, and if the minimum is 0.00 THB and the maximum is 1,165.25 THB, write test cases that cause 0.00 THB and 1,165.25 THB to be deducted. Also, see whether it is possible to invent test cases that might cause a negative deduction or a deduction of more than 1,165.25 THB. Note that it is important to examine the boundaries of the result space because it is not always the case that the boundaries of the input domains represent the same set of circumstances as the boundaries of the output ranges (e.g., consider a sine subroutine). Also, it is not always possible to generate a result outside of the output range; nonetheless, it is worth considering the possibility.

4. Use guideline 2 for each output condition. If an information retrieval system displays the most relevant abstracts based on an input request, but never more than four abstracts, write test cases such that the program displays zero, one, and four abstracts, and write a test case that might cause the program to erroneously display five abstracts.

5. If the input or output of a program is an ordered set (a sequential file, for example, or a linear list or a table), focus attention on the first and last elements of the set.

6. In addition, use your ingenuity to search for other boundary conditions.

1.3.6 Boundary Value Analysis - Example

The triangle analysis program of Week 1 can illustrate the need for boundary value analysis. For the input values to represent a triangle, they must be integers greater than 0 where the sum of any two is greater than the third. If you were defining equivalent partitions, you might define one where this condition is met and another where the sum of two of the integers is not greater than the third. Hence, two possible test cases might be `3-4-5` and `1-2-4`. However, we have missed a likely error. That is, if an expression in the program were coded as `A+B>=C` instead of `A+B>C`, the program would erroneously tell us that `1-2-3` represents a valid scalene triangle. Hence, the important difference between boundary value analysis and equivalence partitioning is that boundary value analysis explores situations on and around the edges of the equivalence partitions.

As an example of a boundary value analysis, consider the following program specification:

> MTEST is a program that grades multiple-choice examinations. The input is a data file named OCR, with multiple records that are 80 characters long. Per the file specification, the first record is a title used as a title on each output report. The next set of records describes the correct answers on the exam. These records contain a "2" as the last character in column 80. In the first record of this set, the number of questions is listed in columns 1-3 (a value of 1-999). Columns 10-59 contain the correct answers for questions 1-50 (any character is valid as an answer). Subsequent records contain, in columns 10-59, the correct answers for questions 51-100, 101-150, and so on.
>
> The third set of records describes the answers of each student; each of these records contains a "3" in column 80. For each student, the first record contains the student's name or number in columns 1- 9 (any characters); columns 10-59 contain the student's answers for questions 1-50. If the test has more than 50 questions, subsequent records for the student contain answers 51-100, 101-150, and so on, in columns 10-59. The maximum number of students is 200. The input data are illustrated in Fig. 1.4. The four output records are:
>
> 1. A report, sorted by student identifier, showing each student's grade (percentage of answers correct) and rank.
> 2. A similar report, but sorted by grade.
> 3. A report indicating the mean, median, and standard deviation of the grades.
> 4. A report, ordered by question number, showing the percentage of students answering each question correctly.

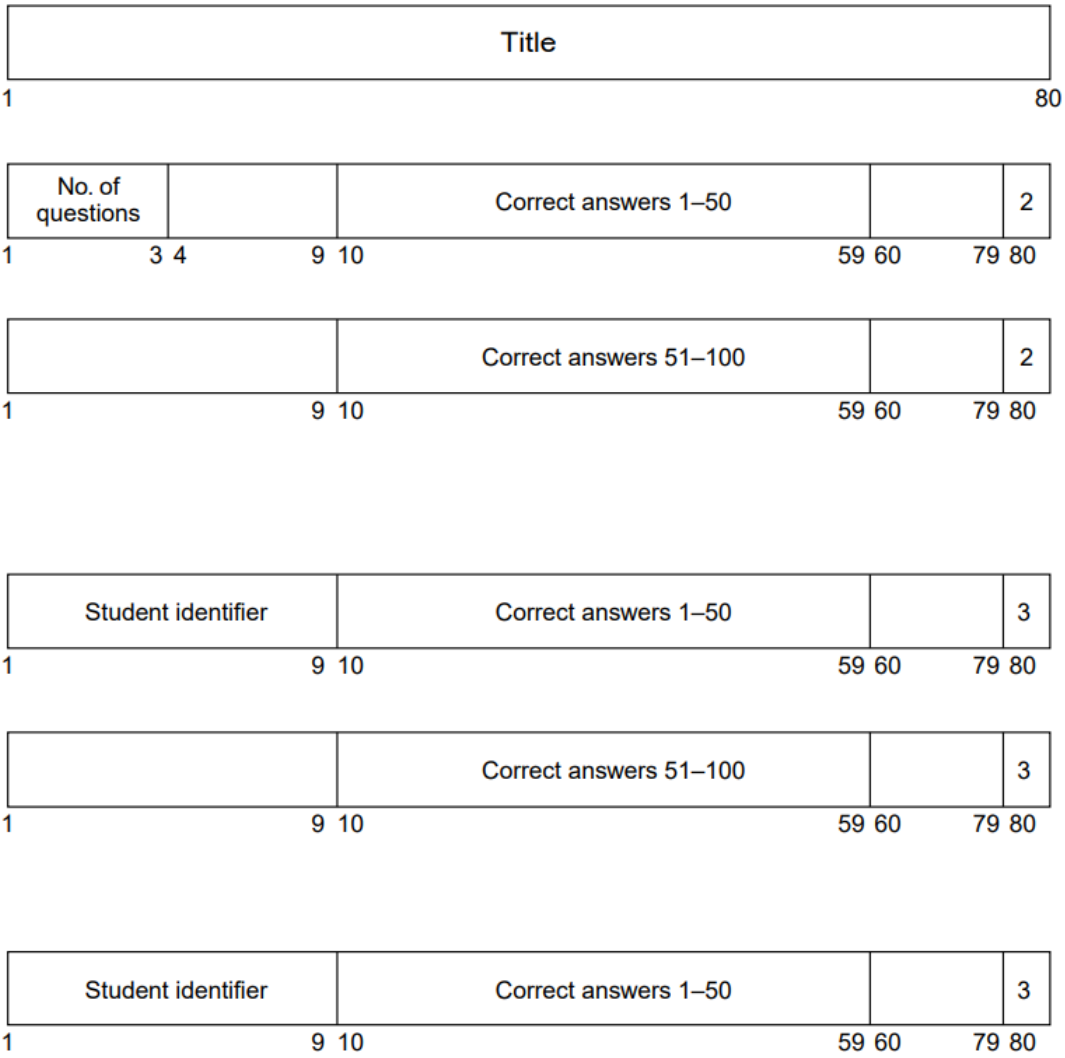


Figure 1.4: Input to the MTEST Program

We can begin by methodically reading the specification, looking for input conditions.

- The first boundary input condition is an empty input file.
- The second input condition is the title record; boundary conditions are a missing title record and the shortest and longest possible titles.
- The next input conditions are the presence of correct-answer records and the number-of-questions field on the first answer record.

  The equivalence class for the number of questions is not 1–999, because something special happens at each multiple of 50 (i.e., multiple records are needed). A reasonable partitioning of this into equivalence classes is 1–50 and 51–999.

  Hence, we need test cases where the number-of-questions field is set to 0, 1, 50, 51, and 999. This covers most of the boundary conditions for the number of correct-answer records; however, three more interesting situations are the absence of answer records and having one too many and one too few answer records (e.g., the number of questions is 60, but there are three answer records in one case and one answer record in the other case).

The unique test cases identified so far are:

1. Empty input file
2. Missing title record
3. 1-character title
4. 80-character title
5. 1-question exam
6. 50-question exam
7. 51-question exam
8. 999-question exam
9. 0-question exam
10. Number-of-questions field with non-numeric value
11. No correct-answer records after title record
12. One too many correct-answer records
13. One too few correct-answer records

- The next input conditions are related to the students' answers. The boundary value test cases here appear to be:

14. 0 students
15. 1 student
16. 200 students
17. 201 students
18. A student has one answer record, but there are two correct-answer records.
19. The above student is the first student in the file.
20. The above student is the last student in the file.
21. A student has two answer records, but there is just one correct answer record.

- You also can derive a useful set of test cases by examining the output boundaries, although some of the output boundaries (e.g., empty report 1) are covered by the existing test cases. The boundary conditions of reports 1 and 2 are:

  - 0 students (same as test 14)
  - 1 student (same as test 15)
  - 200 students (same as test 16)

22. All students receive the same grade.
23. All students receive a different grade.
24. Some, but not all, students receive the same grade (to see if ranks are computed correctly).
25. A student receives a grade of 0.
26. A student receives a grade of 10.

27. A student has the lowest possible identifier value (to check the sort).
28. A student has the highest possible identifier value.
29. The number of students is such that the report is just large enough to fit on one page (to see if an extraneous page is printed).
30. The number of students is such that all students but one fit on one page.

- The boundary conditions from report 3 (mean, median, and standard deviation) are:

  31. The mean is at its maximum (all students have a perfect score).
  32. The mean is 0 (all students receive a grade of 0).
  33. The standard deviation is at its maximum (one student receives a 0 and the other receives a 100).
  34. The standard deviation is 0 (all students receive the same grade).

  Tests 31 and 32 also cover the boundaries of the median. Another useful test case is the situation where there are 0 students (looking for a division by 0 in computing the mean), but this is identical to test case 14.

- An examination of report 4 yields the following boundary value tests:

  35. All students answer question 1 correctly.
  36. All students answer question 1 incorrectly.
  37. All students answer the last question correctly.
  38. All students answer the last question incorrectly.
  39. The number of questions is such that the report is just large enough to fit on one page.
  40. The number of questions is such that all questions but one fit on one page.

An experienced programmer would probably agree at this point that many of these 40 test cases represent common errors that might have been made in developing this program, yet most of these errors probably would go undetected if a random or ad hoc test-case generation method were used. Boundary value analysis, if practiced correctly, is one of the most useful test-case design methods. However, it often is used ineffectively because the technique, on the surface, sounds simple. You should understand that boundary conditions may be very subtle and, hence, identification of them requires a lot of thought.

In next lecture note, we will continue the black-box testing by exploring the "Cause-Effect Graphing" and "Error Guessing" topics.