

## MODULE-02

### FUNCTIONAL TESTING

#### Boundary Value Analysis

We saw that a function maps values from one set (its domain) to values in another set (Its range) and that the domain and range can be cross products of other sets. Any program can be considered to be a function in the sense that program inputs form its domain and program outputs form its range. Input domain testing (also called “boundary value testing”) is the best-known specification-based testing technique. Historically, this form of testing has focused on the input domain; however, it is often a good supplement to apply many of these techniques to develop range-based test cases.

There are two independent considerations that apply to input domain testing.

The first asks whether or not we are concerned with invalid values of variables.

- Normal boundary value testing is concerned only with valid values of the input variables.
- Robust boundary value testing considers invalid and valid variable values.

The second consideration is whether we make the “single fault” assumption common to reliability theory. This assumes that faults are due to incorrect values of a single variable. If this is not warranted, meaning that we are concerned with interaction among two or more variables, we need to take the cross product of the individual variables.

Taken together, the two considerations yield four variations of boundary value testing:

- Normal boundary value testing
- Robust boundary value testing
- Worst-case boundary value testing
- Robust worst-case boundary value testing

For the sake of comprehensible drawings, the discussion relates to a function,  $F$ , of two variables  $x_1$  and  $x_2$ . When the function  $F$  is implemented as a program, the input variables  $x_1$  and  $x_2$  will have some (possibly unstated) boundaries:

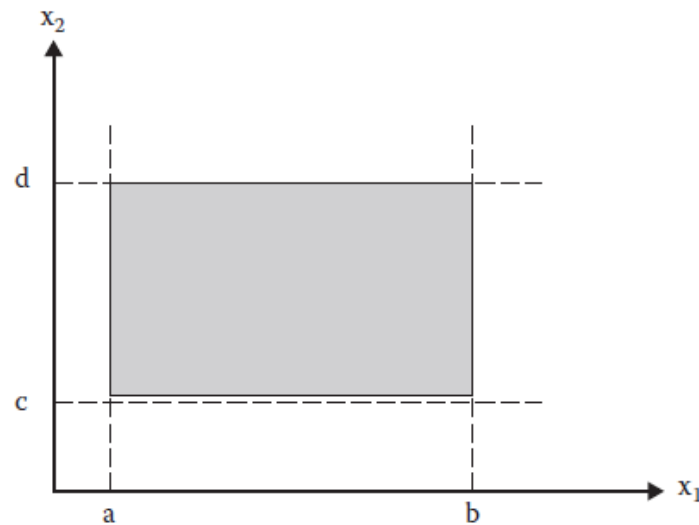
$$a \leq x_1 \leq b$$

$$c \leq x_2 \leq d$$

Unfortunately, the intervals  $[a, b]$  and  $[c, d]$  are referred to as the ranges of  $x_1$  and  $x_2$ , so right away, we have an overloaded term. The intended meaning will always be clear from its context. Strongly typed languages (such as Ada and Pascal) permit explicit definition of such variable ranges. In fact, part of the historical reason for strong typing was to prevent programmers from making the kind of errors that result in faults that are easily revealed by boundary value testing.

Boundary value analysis focuses on the boundary of the input space to identify test cases. The rationale behind boundary value testing is that errors tend to occur near the extreme values of an input variable. The US. Army (CECOM) made a study of its software, and found that a surprising portion of faults turned out to be boundary value faults. Loop conditions. If the cursor is at the last line of a page and new text is added, an anomaly occurs: in the first mode, the new line(s) simply appear, and

the dotted line (page break) is adjusted. In the page display mode, however, the new text is lost — it doesn't appear on either page.



**Figure 2.1** Input Domain of a Function of Two Variables

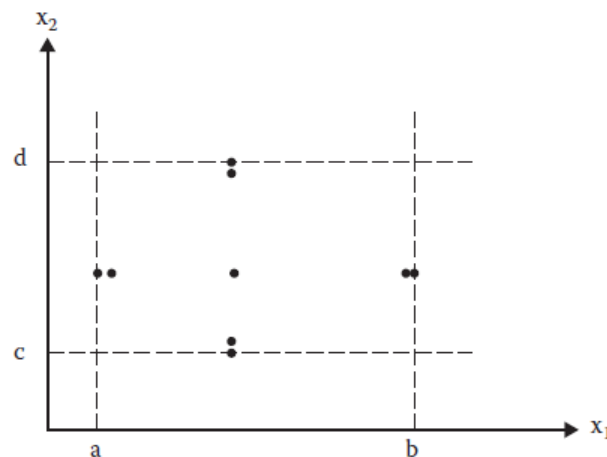
## 2.1 Normal Boundary Value Testing

All four forms of boundary value testing focus on the boundary of the input space to identify test cases. The rationale behind boundary value testing is that errors tend to occur near the Extreme values of an input variable. Loop conditions, for example, may test for  $<$  when they Should test for  $\leq$ , and counters often are “off by one.” (Does counting begin at zero or at one?) The basic idea of boundary value analysis is to use input variable values at their minimum, just Above the minimum, a nominal value, just below their maximum, and at their maximum. A Commercially available testing tool (originally named T) generates such test cases for a properly Specified program. This tool has been successfully integrated with two popular front-end CASE tools (Teamwork from Cadre Systems, and Software through Pictures from Aonix [part of Atego]; for more information, see <http://www.aonix.com/pdf/2140-AON.pdf>). The T tool refers to these values as min, min+, nom, max–, and max. The robust forms add two values, min– and max+.

The next part of boundary value analysis is based on a critical assumption; it is known as the “Single fault” assumption in reliability theory. This says that failures are only rarely the result of the simultaneous occurrence of two (or more) faults. The All Pairs testing approach (described in Chapter 20) contradicts this, with the observation that, in software-controlled medical systems, Almost all faults are the result of interaction between a pair of variables. Thus, the normal and Robust variations cases are obtained by holding the values of all but one variable at their nominal

The basic idea of boundary value analysis is to use input variable values at their minimum, just above the minimum, a nominal value, just below their maximum, and at their maximum. There is a commercially available testing tool (named T) that generates such test cases for a properly specified program. This tool has been successfully integrated with two popular front-end CASE tools (Teamwork from Cadre Systems, and Software Through Pictures from Interactive Development Environments). The T tool refers to these values as min, min+, nom, max- and max; we will use this convention here. The next part of boundary value analysis is based on a critical assumption; it's known as the "single fault" assumption in reliability theory. This says that failures are only rarely the result of the simultaneous occurrence of two (or more) faults. Thus the boundary value analysis test cases are obtained by holding the values of all but one variable at their nominal values, and letting that variable assume its extreme values. The boundary value analysis test cases for our function F of two variables are:

$\{ \langle x1_{nom}, x2_{min} \rangle, \langle x1_{nom}, x2_{min+} \rangle, \langle x1_{nom}, x2_{nom} \rangle, \langle x1_{nom}, x2_{max-} \rangle, \langle x1_{nom}, x2_{max} \rangle, \langle x1_{min}, x2_{nom} \rangle, \langle x1_{min+}, x2_{nom} \rangle, \langle x1_{max-}, x2_{nom} \rangle, \langle x1_{max}, x2_{nom} \rangle \}$



**Figure 2.2 Boundary value analysis test cases for a function of two variables.**

### 2.1.1 Generalizing Boundary Value Analysis

The basic boundary value analysis technique can be generalized in two ways: by the number of variables, and by the kinds of ranges. Generalizing the number of variables is easy: if we have a function of  $n$  variables, we hold all but one at the nominal values, and let the remaining variable assume the min, min+, nom, max- and max values, and repeat this for each variable. Thus for a function of  $n$  variables, boundary value yields  $4n + 1$  test cases. Generalizing ranges depends on the nature (or more precisely, the type) of the variables themselves. In the NextDate function, for example, we have variables for the month, the day, and the year. In a FORTRAN-like language, we would most likely encode these, so that January would correspond to 1, February to 2, and so on. In a language that supports user defined types (like Pascal), we could define the variable month as an enumerated type {Jan., Feb., . . . , Dec.}. Either way, the values for min, min+, nom, max- and max are clear from the context. When a variable has discrete, bounded values, as the

variables in the commission problem have, the min, min+, nom, max- and max are also easily determined. When there are no explicit bounds, as in the triangle problem, we usually have to create “Artificial” bounds. The lower bound of side lengths is clearly 1 (a negative side length is silly), but what might we do for an upper bound? By default, the representable integer (called MAXINT in some languages) is one possibility, or we might impose an arbitrary upper limit such as 200 or 2000.

### 2.1.2 Limitations of Boundary Value Analysis

Boundary value analysis works well when the program to be tested is a function of several independent variables that represent bounded physical quantities. The key words here are *independent* and *physical quantities*. A quick look at the boundary value analysis test cases for NextDate (in section 5.5) shows them to be inadequate. There is very little stress on February and on leap years, for example. The real problem here is that there are interesting dependencies among the month, day, and year variables. Boundary value analysis presumes the variables to be truly independent. Even so, boundary value analysis happens to catch end-of-month and end-of-year faults. Boundary value analysis test cases are derived from the extrema of bounded, independent variables that refer to physical quantities, with no consideration of the nature of the function, nor of the semantic meaning of the variables. We see boundary value analysis test cases to be rudimentary, in the sense that they are obtained with very little insight and imagination. As with so many things, you get what you pay for.

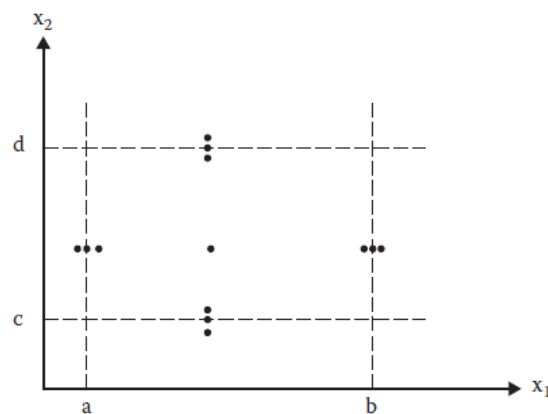
The physical quantity criterion is equally important. When a variable refers to a physical quantity, such as temperature, pressure, air speed, angle of attack, load, and so forth, physical boundaries can be extremely important. (In an interesting example of this, Sky Harbor International Airport in Phoenix had to close on June 26, 1992 because the air temperature was 122 °F. Aircraft pilots were unable to make certain instrument settings before take-off: the instruments could only accept a maximum air temperature of 120 °F.) In another case, a medical analysis system uses stepper motors to position a carousel of samples to be analyzed. It turns out that the mechanics of moving the carousel back to the starting cell often causes the robot arm to miss the first cell. As an example of logical (versus physical) variables, we might look at PINs or telephone numbers. It’s hard to imagine what faults might be revealed by PINs of 0000, 0001, 5000, 9998, and 9999.

## 2.2 Robustness Testing

Robustness testing is a simple extension of boundary analysis: addition to the five boundary value analysis values of a variable, we see what happens when the extrema are exceeded with a value slightly greater than the maximum (max+) and a value slightly less than the minimum (min-). Robustness test cases for our continuing example are shown in Figure 5.3.

Most of the discussion of boundary value analysis applies directly to robustness testing, especially the generalizations and limitations. The most interesting part of robustness testing is not with the inputs, but with the expected outputs. What happens when a physical quantity exceeds its maximum?

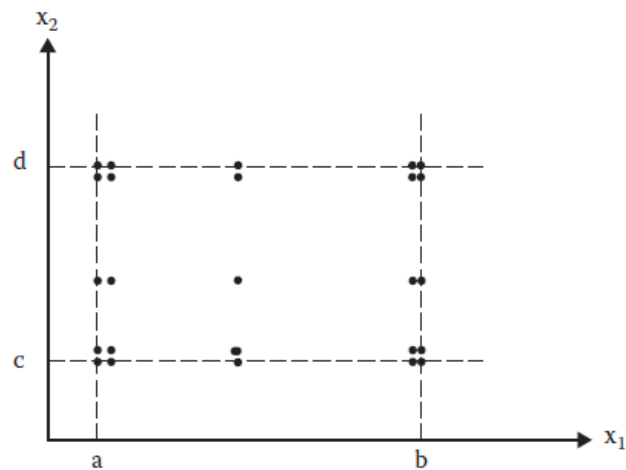
If it is the angle of attack of an airplane wing, the aircraft might stall. If it's the load capacity of a public elevator, we hope nothing special would happen. If it's a date, like May 32, we would expect an error message. The main value of robustness testing is that it forces attention on exception handling. With strongly typed languages, robustness testing may be very awkward. In Pascal, for example, if a variable is defined to be within a certain range, values outside that range result in run-time errors that abort normal execution. This raises an interesting question of implementation philosophy: is it better to perform explicit range checking and use exception handling to deal with "robust values", or is it better to stay with strong typing? handling choice mandates robustness testing.



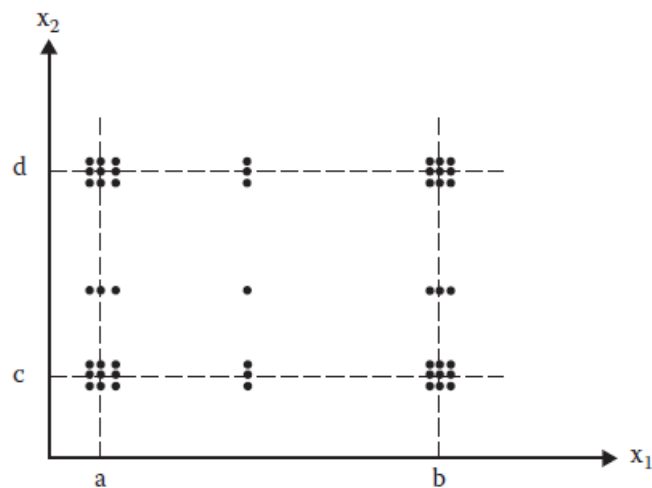
**Figure 2.3 Robustness test cases for a function of two variables.**

### 2.3 Worst Case Testing

Boundary value analysis, as we said earlier, makes the single fault assumption of reliability theory. Rejecting this assumption means that we are interested in what happens more than one variable has an extreme value. In electronic circuit analysis, this is called "worst case analysis"; we use that idea here to generate worst case test cases. For each variable, we start with the five element set that contains the min, min+, nom, max- and max values. We then take the Cartesian product (see Chapter 3) of these sets to generate test cases. The result of the two-variable version of this is shown in Figure 2.4. Worst case testing is clearly more thorough in the sense that boundary value analysis test cases are a proper subset of worst case test cases. It also represents much more effort: worst case testing for a function of  $n$  variables generates  $5n$  test cases, as opposed to  $4n+1$  test cases for boundary value analysis. Worst case testing follows the generalization pattern we saw for boundary value analysis. It also has the same limitations, particularly those related to independence. Probably the best application for worst case testing is where physical variables have numerous interactions, and where failure of the function is extremely costly. For really paranoid testing, we could go to robust worst case testing. This involves the Cartesian product of the seven element sets we used in robustness testing.



**Figure 2.4 Worst-case test cases for a function of two variables.**



**Figure 2.5 Robust worst-case test cases for a function of two variables.**

## 2.4 Special Value Testing

Special value testing is probably the most widely practiced form of functional testing. It also is the most intuitive and the least uniform. Special value testing occurs when a tester uses his/her domain knowledge, experience with similar programs, and information about “soft spots” to devise test cases. We might also call this “ad hoc testing” or “seat of the pants/skirt” testing. There are no guidelines, other than to use “best engineering judgment.” As a result, special value testing is very dependent on the abilities of the tester.

## 2.5 Examples

Each of the three continuing examples is a function of three variables. Printing all the test cases from all the methods for problem is very space consuming, so we'll just have selected examples.

### 2.5.1 Test Cases for the Triangle Problem

In the problem statement, no conditions are specified on the triangle sides, other than being integers. Obviously, the lower bounds of the ranges are all 1. We arbitrarily take 200 as an upper bound. For each side, the test values are {1, 2, 100, 199, 200}. Robust boundary value test cases will add {0, 201}. Table 5.1 contains boundary value test cases using these ranges. Notice that test cases 3, 8, and 13 are identical; two should be deleted. Further, there is no test case for scalene triangles.

The cross-product of test values will have 125 test cases (some of which will be repeated)—too many to list here. The full set is available as a spreadsheet in the set of student exercises. Table 2.2 only lists the first 25 worst-case boundary value test cases for the triangle problem. You can picture them as a plane slice through the cube (actually it is a rectangular parallelepiped) in which  $a = 1$  and the other two variables take on their full set of cross-product values.

**Table 2.1 Normal Boundary Value Test Cases**

| Case | a   | b   | c   | <i>Expected Output</i> |
|------|-----|-----|-----|------------------------|
| 1    | 100 | 100 | 1   | Isosceles              |
| 2    | 100 | 100 | 2   | Isosceles              |
| 3    | 100 | 100 | 100 | Equilateral            |
| 4    | 100 | 100 | 199 | Isosceles              |
| 5    | 100 | 100 | 200 | Not a triangle         |
| 6    | 100 | 1   | 100 | Isosceles              |
| 7    | 100 | 2   | 100 | Isosceles              |
| 8    | 100 | 100 | 100 | Equilateral            |
| 9    | 100 | 199 | 100 | Isosceles              |
| 10   | 100 | 200 | 100 | Not a triangle         |
| 11   | 1   | 100 | 100 | Isosceles              |
| 12   | 2   | 100 | 100 | Isosceles              |
| 13   | 100 | 100 | 100 | Equilateral            |
| 14   | 199 | 100 | 100 | Isosceles              |
| 15   | 200 | 100 | 100 | Not a triangle         |

**Table 2.2 (Selected) Worst-Case Boundary Value Test Cases**

| Case | a | b   | c   | <i>Expected Output</i> |
|------|---|-----|-----|------------------------|
| 1    | 1 | 1   | 1   | Equilateral            |
| 2    | 1 | 1   | 2   | Not a triangle         |
| 3    | 1 | 1   | 100 | Not a triangle         |
| 4    | 1 | 1   | 199 | Not a triangle         |
| 5    | 1 | 1   | 200 | Not a triangle         |
| 6    | 1 | 2   | 1   | Not a triangle         |
| 7    | 1 | 2   | 2   | Isosceles              |
| 8    | 1 | 2   | 100 | Not a triangle         |
| 9    | 1 | 2   | 199 | Not a triangle         |
| 10   | 1 | 2   | 200 | Not a triangle         |
| 11   | 1 | 100 | 1   | Not a triangle         |
| 12   | 1 | 100 | 2   | Not a triangle         |
| 13   | 1 | 100 | 100 | Isosceles              |
| 14   | 1 | 100 | 199 | Not a triangle         |
| 15   | 1 | 100 | 200 | Not a triangle         |
| 16   | 1 | 199 | 1   | Not a triangle         |
| 17   | 1 | 199 | 2   | Not a triangle         |
| 18   | 1 | 199 | 100 | Not a triangle         |
| 19   | 1 | 199 | 199 | Isosceles              |
| 20   | 1 | 199 | 200 | Not a triangle         |
| 21   | 1 | 200 | 1   | Not a triangle         |
| 22   | 1 | 200 | 2   | Not a triangle         |
| 23   | 1 | 200 | 100 | Not a triangle         |
| 24   | 1 | 200 | 199 | Not a triangle         |
| 25   | 1 | 200 | 200 | Isosceles              |

### 2.5.2 Test Cases for the NextDate Function

All 125 worst-case test cases for NextDate are listed in Table 5.3. Take some time to examine it for gaps of untested functionality and for redundant testing. For example, would anyone actually want to test January 1 in five different years? Is the end of February tested sufficiently?



| <i>Case</i> | <i>Month</i> | <i>Day</i> | <i>Year</i> | <i>Expected Output</i> |
|-------------|--------------|------------|-------------|------------------------|
| 1           | 1            | 1          | 1812        | 1, 2, 1812             |
| 2           | 1            | 1          | 1813        | 1, 2, 1813             |
| 3           | 1            | 1          | 1912        | 1, 2, 1912             |
| 4           | 1            | 1          | 2011        | 1, 2, 2011             |
| 5           | 1            | 1          | 2012        | 1, 2, 2012             |
| 6           | 1            | 2          | 1812        | 1, 3, 1812             |
| 7           | 1            | 2          | 1813        | 1, 3, 1813             |
| 8           | 1            | 2          | 1912        | 1, 3, 1912             |
| 9           | 1            | 2          | 2011        | 1, 3, 2011             |
| 10          | 1            | 2          | 2012        | 1, 3, 2012             |
| 11          | 1            | 15         | 1812        | 1, 16, 1812            |
| 12          | 1            | 15         | 1813        | 1, 16, 1813            |
| 13          | 1            | 15         | 1912        | 1, 16, 1912            |
| 14          | 1            | 15         | 2011        | 1, 16, 2011            |
| 15          | 1            | 15         | 2012        | 1, 16, 2012            |
| 16          | 1            | 30         | 1812        | 1, 31, 1812            |
| 17          | 1            | 30         | 1813        | 1, 31, 1813            |
| 18          | 1            | 30         | 1912        | 1, 31, 1912            |
| 19          | 1            | 30         | 2011        | 1, 31, 2011            |
| 20          | 1            | 30         | 2012        | 1, 31, 2012            |
| 21          | 1            | 31         | 1812        | 2, 1, 1812             |
| 22          | 1            | 31         | 1813        | 2, 1, 1813             |
| 23          | 1            | 31         | 1912        | 2, 1, 1912             |
| 24          | 1            | 31         | 2011        | 2, 1, 2011             |
| 25          | 1            | 31         | 2012        | 2, 1, 2012             |
| 26          | 2            | 1          | 1812        | 2, 2, 1812             |
| 27          | 2            | 1          | 1813        | 2, 2, 1813             |
| 28          | 2            | 1          | 1912        | 2, 2, 1912             |

(continued)

**Table 2.3 Worst-Case Test Cases**

| Case | Month | Day | Year | Expected Output |
|------|-------|-----|------|-----------------|
| 29   | 2     | 1   | 2011 | 2, 2, 2011      |
| 30   | 2     | 1   | 2012 | 2, 2, 2012      |
| 31   | 2     | 2   | 1812 | 2, 3, 1812      |
| 32   | 2     | 2   | 1813 | 2, 3, 1813      |
| 33   | 2     | 2   | 1912 | 2, 3, 1912      |
| 34   | 2     | 2   | 2011 | 2, 3, 2011      |
| 35   | 2     | 2   | 2012 | 2, 3, 2012      |
| 36   | 2     | 15  | 1812 | 2, 16, 1812     |
| 37   | 2     | 15  | 1813 | 2, 16, 1813     |
| 38   | 2     | 15  | 1912 | 2, 16, 1912     |
| 39   | 2     | 15  | 2011 | 2, 16, 2011     |
| 40   | 2     | 15  | 2012 | 2, 16, 2012     |
| 41   | 2     | 30  | 1812 | Invalid date    |
| 42   | 2     | 30  | 1813 | Invalid date    |
| 43   | 2     | 30  | 1912 | Invalid date    |
| 44   | 2     | 30  | 2011 | Invalid date    |
| 45   | 2     | 30  | 2012 | Invalid date    |
| 46   | 2     | 31  | 1812 | Invalid date    |
| 47   | 2     | 31  | 1813 | Invalid date    |
| 48   | 2     | 31  | 1912 | Invalid date    |
| 49   | 2     | 31  | 2011 | Invalid date    |
| 50   | 2     | 31  | 2012 | Invalid date    |
| 51   | 6     | 1   | 1812 | 6, 2, 1812      |
| 52   | 6     | 1   | 1813 | 6, 2, 1813      |
| 53   | 6     | 1   | 1912 | 6, 2, 1912      |
| 54   | 6     | 1   | 2011 | 6, 2, 2011      |
| 55   | 6     | 1   | 2012 | 6, 2, 2012      |
| 56   | 6     | 2   | 1812 | 6, 3, 1812      |
| 57   | 6     | 2   | 1813 | 6, 3, 1813      |

(continued)

| Case | Month | Day | Year | Expected Output |
|------|-------|-----|------|-----------------|
| 87   | 11    | 15  | 1813 | 11, 16, 1813    |
| 88   | 11    | 15  | 1912 | 11, 16, 1912    |
| 89   | 11    | 15  | 2011 | 11, 16, 2011    |
| 90   | 11    | 15  | 2012 | 11, 16, 2012    |
| 91   | 11    | 30  | 1812 | 12, 1, 1812     |
| 92   | 11    | 30  | 1813 | 12, 1, 1813     |
| 93   | 11    | 30  | 1912 | 12, 1, 1912     |
| 94   | 11    | 30  | 2011 | 12, 1, 2011     |
| 95   | 11    | 30  | 2012 | 12, 1, 2012     |
| 96   | 11    | 31  | 1812 | Invalid date    |
| 97   | 11    | 31  | 1813 | Invalid date    |
| 98   | 11    | 31  | 1912 | Invalid date    |
| 99   | 11    | 31  | 2011 | Invalid date    |
| 100  | 11    | 31  | 2012 | Invalid date    |
| 101  | 12    | 1   | 1812 | 12, 2, 1812     |
| 102  | 12    | 1   | 1813 | 12, 2, 1813     |
| 103  | 12    | 1   | 1912 | 12, 2, 1912     |
| 104  | 12    | 1   | 2011 | 12, 2, 2011     |
| 105  | 12    | 1   | 2012 | 12, 2, 2012     |
| 106  | 12    | 2   | 1812 | 12, 3, 1812     |
| 107  | 12    | 2   | 1813 | 12, 3, 1813     |
| 108  | 12    | 2   | 1912 | 12, 3, 1912     |
| 109  | 12    | 2   | 2011 | 12, 3, 2011     |
| 110  | 12    | 2   | 2012 | 12, 3, 2012     |
| 111  | 12    | 15  | 1812 | 12, 16, 1812    |
| 112  | 12    | 15  | 1813 | 12, 16, 1813    |
| 113  | 12    | 15  | 1912 | 12, 16, 1912    |
| 114  | 12    | 15  | 2011 | 12, 16, 2011    |
| 115  | 12    | 15  | 2012 | 12, 16, 2012    |

(continued)

| Case | Month | Day | Year | Expected Output |
|------|-------|-----|------|-----------------|
| 58   | 6     | 2   | 1912 | 6, 3, 1912      |
| 59   | 6     | 2   | 2011 | 6, 3, 2011      |
| 60   | 6     | 2   | 2012 | 6, 3, 2012      |
| 61   | 6     | 15  | 1812 | 6, 16, 1812     |
| 62   | 6     | 15  | 1813 | 6, 16, 1813     |
| 63   | 6     | 15  | 1912 | 6, 16, 1912     |
| 64   | 6     | 15  | 2011 | 6, 16, 2011     |
| 65   | 6     | 15  | 2012 | 6, 16, 2012     |
| 66   | 6     | 30  | 1812 | 7, 1, 1812      |
| 67   | 6     | 30  | 1813 | 7, 1, 1813      |
| 68   | 6     | 30  | 1912 | 7, 1, 1912      |
| 69   | 6     | 30  | 2011 | 7, 1, 2011      |
| 70   | 6     | 30  | 2012 | 7, 1, 2012      |
| 71   | 6     | 31  | 1812 | Invalid date    |
| 72   | 6     | 31  | 1813 | Invalid date    |
| 73   | 6     | 31  | 1912 | Invalid date    |
| 74   | 6     | 31  | 2011 | Invalid date    |
| 75   | 6     | 31  | 2012 | Invalid date    |
| 76   | 11    | 1   | 1812 | 11, 2, 1812     |
| 77   | 11    | 1   | 1813 | 11, 2, 1813     |
| 78   | 11    | 1   | 1912 | 11, 2, 1912     |
| 79   | 11    | 1   | 2011 | 11, 2, 2011     |
| 80   | 11    | 1   | 2012 | 11, 2, 2012     |
| 81   | 11    | 2   | 1812 | 11, 3, 1812     |
| 82   | 11    | 2   | 1813 | 11, 3, 1813     |
| 83   | 11    | 2   | 1912 | 11, 3, 1912     |
| 84   | 11    | 2   | 2011 | 11, 3, 2011     |
| 85   | 11    | 2   | 2012 | 11, 3, 2012     |
| 86   | 11    | 15  | 1812 | 11, 16, 1812    |

(continued)

| Case | Month | Day | Year | Expected Output |
|------|-------|-----|------|-----------------|
| 116  | 12    | 30  | 1812 | 12, 31, 1812    |
| 117  | 12    | 30  | 1813 | 12, 31, 1813    |
| 118  | 12    | 30  | 1912 | 12, 31, 1912    |
| 119  | 12    | 30  | 2011 | 12, 31, 2011    |
| 120  | 12    | 30  | 2012 | 12, 31, 2012    |
| 121  | 12    | 31  | 1812 | 1, 1, 1813      |
| 122  | 12    | 31  | 1813 | 1, 1, 1814      |
| 123  | 12    | 31  | 1912 | 1, 1, 1913      |
| 124  | 12    | 31  | 2011 | 1, 1, 2012      |
| 125  | 12    | 31  | 2012 | 1, 1, 2013      |

### 2.5.3 Test Cases for the Commission Problem

Instead of going through 125 boring test cases again, we will look at some more interesting test cases for the commission problem. This time, we will look at boundary values derived from the Output range, especially near the threshold points of \$1000 and \$1800 where the commission percentage changes. The output space of the commission is shown in Figure 2.6. The intercepts of these threshold planes with the axes are shown.

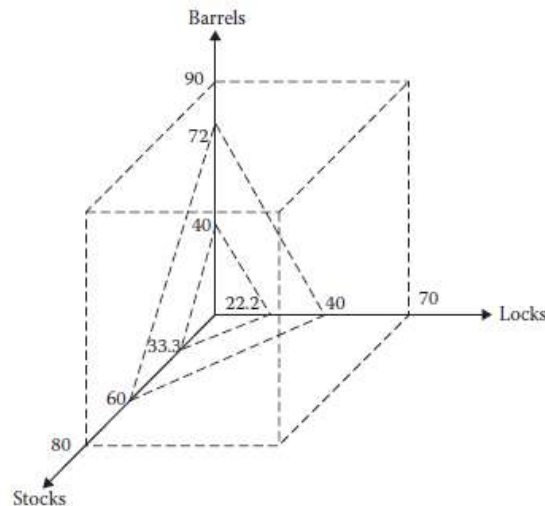


Figure 2.6 Input space of the commission problem.

Table 2.4 Output Boundary Value Analysis Test Cases

| Case | Locks | Stocks | Barrels | Sales | Comm   | Comment          |
|------|-------|--------|---------|-------|--------|------------------|
| 1    | 1     | 1      | 1       | 100   | 10     | Output minimum   |
| 2    | 1     | 1      | 2       | 125   | 12.5   | Output minimum + |
| 3    | 1     | 2      | 1       | 130   | 13     | Output minimum + |
| 4    | 2     | 1      | 1       | 145   | 14.5   | Output minimum + |
| 5    | 5     | 5      | 5       | 500   | 50     | Midpoint         |
| 6    | 10    | 10     | 9       | 975   | 97.5   | Border point –   |
| 7    | 10    | 9      | 10      | 970   | 97     | Border point –   |
| 8    | 9     | 10     | 10      | 955   | 95.5   | Border point –   |
| 9    | 10    | 10     | 10      | 1000  | 100    | Border point     |
| 10   | 10    | 10     | 11      | 1025  | 103.75 | Border point +   |
| 11   | 10    | 11     | 10      | 1030  | 104.5  | Border point +   |
| 12   | 11    | 10     | 10      | 1045  | 106.75 | Border point +   |
| 13   | 14    | 14     | 14      | 1400  | 160    | Midpoint         |
| 14   | 18    | 18     | 17      | 1775  | 216.25 | Border point –   |
| 15   | 18    | 17     | 18      | 1770  | 215.5  | Border point –   |
| 16   | 17    | 18     | 18      | 1755  | 213.25 | Border point –   |
| 17   | 18    | 18     | 18      | 1800  | 220    | Border point     |
| 18   | 18    | 18     | 19      | 1825  | 225    | Border point +   |
| 19   | 18    | 19     | 18      | 1830  | 226    | Border point +   |
| 20   | 19    | 18     | 18      | 1845  | 229    | Border point +   |
| 21   | 48    | 48     | 48      | 4800  | 820    | Midpoint         |
| 22   | 70    | 80     | 89      | 7775  | 1415   | Output maximum – |
| 23   | 70    | 79     | 90      | 7770  | 1414   | Output maximum – |
| 24   | 69    | 80     | 90      | 7755  | 1411   | Output maximum – |
| 25   | 70    | 80     | 90      | 7800  | 1420   | Output maximum   |

The volume between the origin and the lower plane corresponds to sales below the \$1000 threshold. The volume between the two planes is the 15% commission range. Part of the reason for using the output range to determine test cases is that cases from the input range are almost all in the 20% zone. We want to find input variable combinations that stress the sales/commission boundary values: \$100, \$1000, \$1800, and \$7800. The minimum and maximum were easy, and

**Table 2.5 Output Special Value Test Cases**

| Case | Locks | Stocks | Barrels | Sales | Comm   | Comment        |
|------|-------|--------|---------|-------|--------|----------------|
| 1    | 10    | 11     | 9       | 1005  | 100.75 | Border point + |
| 2    | 18    | 17     | 19      | 1795  | 219.25 | Border point - |
| 3    | 18    | 19     | 17      | 1805  | 221    | Border point + |

the numbers happen to work out so that the border points are easy to generate. Here is where it gets interesting: test case 9 is the \$1000 border point. If we tweak the input variables, we get values just below and just above the border (cases 6–8 and 10–12). If we wanted to, we could pick values near the borders such as (22, 1, 1). As we continue in this way, we have a sense that we are “exercising” Interesting parts of the code. We might claim that this is really a form of special value testing because we used our mathematical insight to generate test cases.

Table 2.4 contains test cases derived from boundary values on the output side of the commission function.

## 2.6 Guidelines for Boundary value Testing

With the exception of special value testing, the test methods based on the boundary values of a function (program) are the most rudimentary of all functional testing methods. They share the common assumption that the input variables are truly independent, and when this assumption is not warranted, the methods generate unsatisfactory test cases (such as February 31, 1912 for NextDate). These methods have two other distinctions: normal versus robust values, and the single fault versus the multiple fault assumption. Just using these distinctions carefully will result in better testing.

Each of these methods can be applied to the output range of a program, as we did for the commission problem.

Another useful form of output-based test cases is for systems that generate error messages. The tester should devise test cases to check that error messages are generated when they are appropriate, and are not falsely generated. Boundary value analysis can also be used for internal variables, such as loop control variables, indices, and pointers. Strictly speaking, these are not input variables, but errors in the use of these variables are quite common. Robustness testing is a good choice for testing internal variables.

## Equivalence Classes

We noted that the important aspect of equivalence classes is that they form a partition of a set, where partition refers to a collection of mutually disjoint subsets whose union is the entire set. This has two important implications for testing: the fact that the entire set is represented provides a form of completeness, and the disjointness assures a form of non-redundancy. Because the subsets are determined by an equivalence relation, the elements of a subset have something in common. The idea of equivalence class testing is to identify test cases by using one element from each equivalence class. If the equivalence classes are chosen wisely, this greatly reduces the potential redundancy among test cases. In the Triangle Problem, for example, we would certainly have a test case for an equilateral triangle, and we might pick the triple (5, 5, 5) as inputs for a test case. If we did this, we would not expect to learn much from test cases such as (6, 6, 6) and (100, 100, 100). Our intuition tells us that these would be “treated the same” as the first test case, thus they would be redundant. When we consider structural testing in Part III, we shall see that “treated the same” maps onto “traversing the same execution path”.

The key (and the craft!) of equivalence class testing is the choice of the equivalence relation that determines the classes. Very often, we make this choice by “second guessing” the likely implementation, and thinking about the functional manipulations that must somehow be present in the implementation. We will illustrate this with our continuing examples, but first, we need to make a distinction between weak and strong equivalence class testing. After that, we will compare these to the traditional form of equivalence class testing.

Suppose our program is a function of three variables, a, b, and c, and the input domain consists of sets A, B, and C. Now, suppose we choose an “appropriate” equivalence relation, which induces the following partition:

### 2.7 Traditional Equivalence Class Testing

Most of the standard testing texts (e.g., Myers, 1979; Mosley, 1993) discuss equivalence classes based on valid and invalid variable values. Traditional equivalence class testing is nearly identical to weak robust equivalence class testing (see Section 6.3.3). This traditional form focuses on invalid data values, and it is/was a consequence of the dominant style of programming in the 1960s and 1970s. Input data validation was an important issue at the time, and “Garbage In, Garbage Out” was the programmer’s watchword. In the early years, it was the program user’s responsibility to provide valid data. There was no guarantee about results based on invalid data. The term soon became known as GIGO. The usual response to GIGO was extensive input validation sections of a program. Authors and seminar leaders frequently commented that, in the classic afferent/central/efferent architecture of structured programming, the afferent portion often represented 80% of the total source code. In this context, it is natural to emphasize input data validation. Clearly, the defense against GIGO was to have extensive testing to assure data validity.

The gradual shift to modern programming languages, especially those that feature strong data typing, and then to graphical user interfaces (GUIs) obviated much of the need for input data validation. Indeed, good

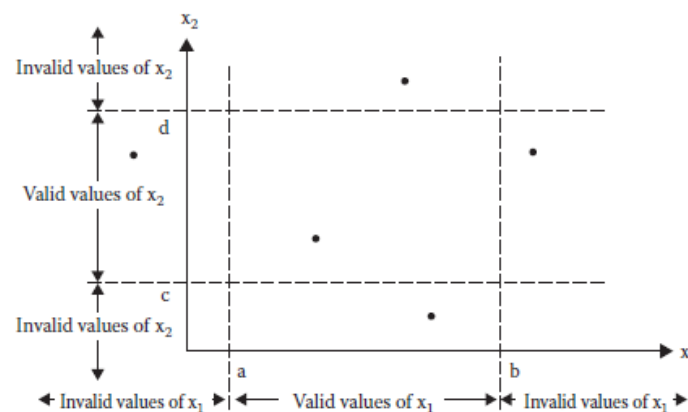
use of user interface devices such as drop-down lists and slider bars reduces the likelihood of bad input data.

Traditional equivalence class testing echoes the process of boundary value testing. Figure 2.7 shows test cases for a function  $F$  of two variables  $x_1$  and  $x_2$ , as we had in Chapter 5. The extension to more realistic cases of  $n$  variables proceeds as follows:

1. Test  $F$  for valid values of all variables.
2. If step 1 is successful, then test  $F$  for invalid values of  $x_1$  with valid values of the remaining variables. Any failure will be due to a problem with an invalid value of  $x_1$ .
3. Repeat step 2 for the remaining variables.

One clear advantage of this process is that it focuses on finding faults due to invalid data. Since the GIGO concern was on invalid data, the kinds of combinations that we saw in the worst-case variations of boundary value testing were ignored.

Figure 2.7 shows the five test cases for this process for our continuing function  $F$  of two variables.



**Figure 2.7 Traditional equivalence class test cases.**

## 2.8 Improved Equivalence Class Testing

The key (and the craft!) of equivalence class testing is the choice of the equivalence relation that determines the classes. Very often, we make this choice by second-guessing the likely implementation and thinking about the functional manipulations that must somehow be present in the implementation. We will illustrate this with our continuing examples. We need to enrich the function we used in boundary value testing. Again, for the sake of comprehensible drawings, the discussion relates to a function,  $F$ , of two variables  $x_1$  and  $x_2$ . When  $F$  is implemented as a program, the input variables  $x_1$  and  $x_2$  will have the following boundaries, and intervals within the boundaries:

$$\begin{aligned} & a \leq x_1 \leq d, \text{ with intervals } [a, b), [b, c), [c, d] \\ & e \leq x_2 \leq g, \text{ with intervals } [e, f), [f, g] \end{aligned}$$

where square brackets and parentheses denote, respectively, closed and open interval endpoints. The intervals presumably correspond to some distinction in the program being tested, for example, the commission ranges in the commission problem. These ranges are equivalence classes. Invalid values of  $x_1$  and  $x_2$  are  $x_1 < a$ ,  $x_1 > d$ , and  $x_2 < e$ ,  $x_2 > g$ .

The equivalence classes of valid values are

$V1 = \{x1: a \leq x1 < b\}$ ,  $V2 = \{x1: b \leq x1 < c\}$ ,  $V3 = \{x1: c \leq x1 \leq d\}$ ,  $V4 = \{x2: e \leq x2 < f\}$ ,  $V5 = \{x2: f \leq x2 \leq g\}$

The equivalence classes of invalid values are

$NV1 = \{x1: x1 < a\}$ ,  $NV2 = \{x1: d < x1\}$ ,  $NV3 = \{x2: x2 < e\}$ ,  $NV4 = \{x2: g < x2\}$

The equivalence classes V1, V2, V3, V4, V5, NV1, NV2, NV3, and NV4 are disjoint, and their union is the entire plane. In the succeeding discussions, we will just use the interval notation rather than the full formal set definition. Types of Equivalence Class Partitioning

- i. Weak Normal Equivalence Class Testing.
- ii. Strong Normal Equivalence Class Testing.
- iii. Weak Robust Equivalence Class Testing.
- iv. Strong Robust Equivalence Class Testing.

### 2.8.1 Weak Normal Equivalence Class Testing

With the notation as given previously, weak normal equivalence class testing is accomplished by using one variable from each equivalence class (interval) in a test case. (Note the effect of the single fault assumption.) For the running example, we would end up with the three weak equivalence class test cases shown in Figure 2.8. This figure will be repeated for the remaining forms of equivalence class testing, but, for clarity, without the indication of valid and invalid ranges. These three test cases use one value from each equivalence class. The test case in the lower left rectangle corresponds to a value of  $x1$  in the class  $[a, b)$ , and to a value of  $x2$  in the class  $[e, f)$ . The test case in the upper center rectangle corresponds to a value of  $x1$  in the class  $[b, c)$  and to a value of  $x2$  in the class  $[f, g]$ . The third test case could be in either rectangle on the right side of the valid values. We identified these in a systematic way, thus the apparent pattern. In fact, we will always have the same number of weak equivalence class test cases as classes in the partition with the largest number of subsets. What can we learn from a weak normal equivalence class test case that fails, that is, one for which the expected and actual outputs are inconsistent? There could be a problem with  $x1$ , or a problem with  $x2$ , or maybe an interaction between the two. This ambiguity is the reason for the “weak” designation. If the expectation of failure is low, as it is for regression testing, this can be an acceptable choice. When more fault isolation is required, the stronger forms, discussed next, are indicated.

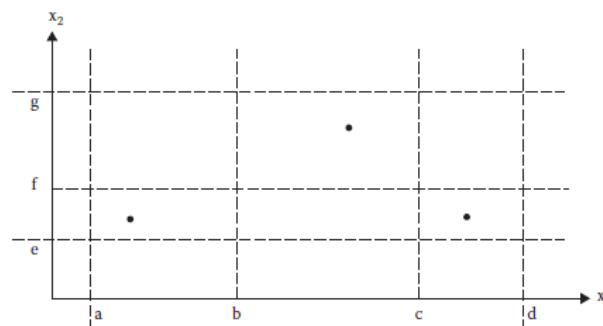


Figure 2.8 Weak normal equivalence class test cases.

### 2.8.2 Strong Normal Equivalence Class Testing

Strong equivalence class testing is based on the multiple fault assumption, so we need test cases from each element of the Cartesian product of the equivalence classes, as shown in Figure 2.9. Notice the similarity between the pattern of these test cases and the construction of a truth table in propositional logic. The Cartesian product guarantees that we have a notion of “completeness” in two senses: we cover all the equivalence classes, and we have one of each possible combination of inputs. As we shall see from our continuing examples, the key to “good” equivalence class testing is the selection of the equivalence relation. Watch for the notion of inputs being “treated the same.” Most of the time, equivalence class testing defines classes of the input domain. There is no reason why we could not define equivalence relations on the output range of the program function being tested; in fact, this is the simplest approach for the triangle problem.

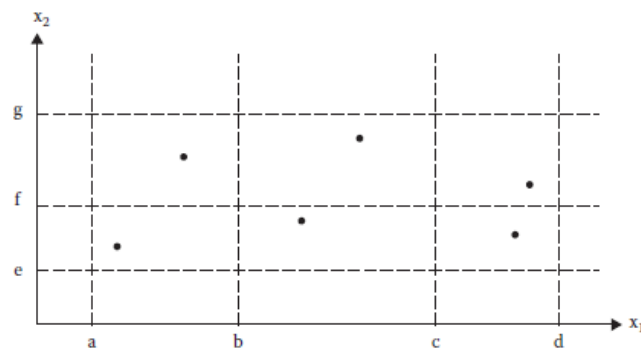


Figure 2.9 Strong normal equivalence class test cases.

### 2.8.3 Weak Robust Equivalence Class Testing

The name for this form is admittedly counterintuitive and oxymoronic. How can something be both weak and robust? The robust part comes from consideration of invalid values, and the weak part refers to the single fault assumption. The process of weak robust equivalence class testing is a simple extension of that for weak normal equivalence class testing—pick test cases such that each equivalence class is represented. In Figure 2.10, the test cases for valid classes are as those in Figure 2.8. The two additional test cases cover all four classes of invalid values. The process is similar to that for boundary value testing:

1. For valid inputs, use one value from each valid class (as in what we have called weak normal equivalence class testing). (Note that each input in these test cases will be valid.)
2. For invalid inputs, a test case will have one invalid value and the remaining values will all be valid. (Thus, a “single failure” should cause the test case to fail.)

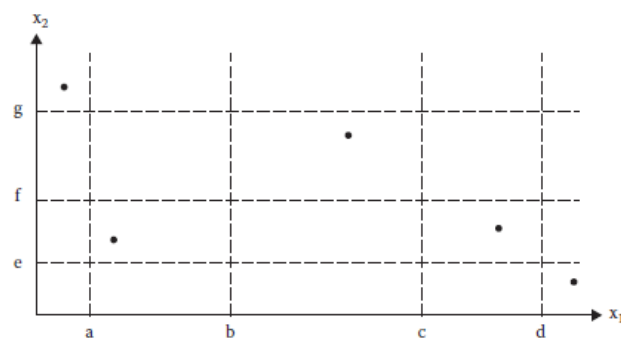
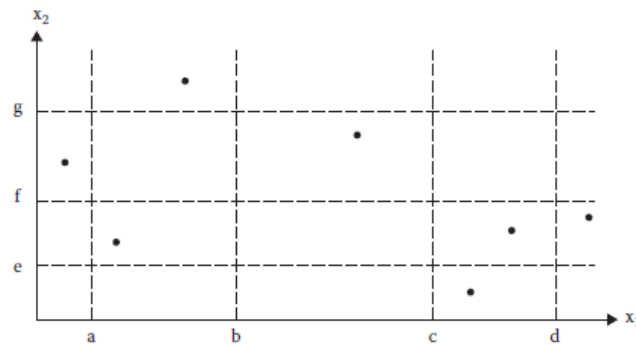


Figure 2.10 Weak robust equivalence class test cases.





**Figure 2.11 Revised weak robust equivalence class test cases.**

The test cases resulting from this strategy are shown in Figure 2.10. There is a potential problem with these test cases. Consider the test cases in the upper left and lower right corners. Each of the test cases represents values from two invalid equivalence classes. Failure of either of these could be due to the interaction of two variables. Figure 2.11 presents a compromise between “pure” weak normal equivalence class testing and its robust extension.

## 2.9 Equivalence Class Test Cases for the Triangle Problem

In the problem statement, we note that four possible outputs can occur: NotATriangle, Scalene, Isosceles, and Equilateral. We can use these to identify output (range) equivalence classes as follows.

R1 = {<a, b, c>: the triangle with sides a, b, and c is equilateral}

R2 = {<a, b, c>: the triangle with sides a, b, and c is isosceles}

R3 = {<a, b, c>: the triangle with sides a, b, and c is scalene}

R4 = {<a, b, c>: sides a, b, and c do not form a triangle}

Four weak normal equivalence class test cases, chosen arbitrarily from each class are as follows:

| <i>Test Case</i> | <i>a</i> | <i>b</i> | <i>c</i> | <i>Expected Output</i> |
|------------------|----------|----------|----------|------------------------|
| WN1              | 5        | 5        | 5        | Equilateral            |
| WN2              | 2        | 2        | 3        | Isosceles              |
| WN3              | 3        | 4        | 5        | Scalene                |
| WN4              | 4        | 1        | 2        | Not a triangle         |

Because no valid subintervals of variables a, b, and c exist, the strong normal equivalence class test cases are identical to the weak normal equivalence class test cases.

Considering the invalid values for a, b, and c yields the following additional weak robust equivalence class test cases. (The invalid values could be zero, any negative number, or any number greater than 200.)

| Test Case | a   | b   | c   | Expected Output                                    |
|-----------|-----|-----|-----|--|
| WR1       | -1  | 5   | 5   | Value of a is not in the range of permitted values |
| WR2       | 5   | -1  | 5   | Value of b is not in the range of permitted values |
| WR3       | 5   | 5   | -1  | Value of c is not in the range of permitted values |
| WR4       | 201 | 5   | 5   | Value of a is not in the range of permitted values |
| WR5       | 5   | 201 | 5   | Value of b is not in the range of permitted values |
| WR6       | 5   | 5   | 201 | Value of c is not in the range of permitted values |

Here is one “corner” of the cube in three-space of the additional strong robust equivalence class test cases:

| Test Case | a  | b  | c  | Expected Output  |
|-----------|----|----|----|--|
| SR1       | -1 | 5  | 5  | Value of a is not in the range of permitted values         |
| SR2       | 5  | -1 | 5  | Value of b is not in the range of permitted values         |
| SR3       | 5  | 5  | -1 | Value of c is not in the range of permitted values         |
| SR4       | -1 | -1 | 5  | Values of a, b are not in the range of permitted values    |
| SR5       | 5  | -1 | -1 | Values of b, c are not in the range of permitted values    |
| SR6       | -1 | 5  | -1 | Values of a, c are not in the range of permitted values    |
| SR7       | -1 | -1 | -1 | Values of a, b, c are not in the range of permitted values |

Notice how thoroughly the expected outputs describe the invalid input values.

Equivalence class testing is clearly sensitive to the equivalence relation used to define classes.

Here is another instance of craftsmanship. If we base equivalence classes on the output domain, we obtain a richer set of test cases. What are some of the possibilities for the three integers, a, b, and c? They can all be equal, exactly one pair can be equal (this can happen in three ways), or none can be equal.

**D1 = {<a, b, c>: a = b = c}**

**D2 = {<a, b, c>: a = b, a ≠ c}**

**D3 = {<a, b, c>: a = c, a ≠ b}**

**D4 = {<a, b, c>: b = c, a ≠ b}**

**D5 = {<a, b, c>: a ≠ b, a ≠ c, b ≠ c}**

As a separate question, we can apply the triangle property to see if they even constitute a triangle. (For example, the triplet <1, 4, 1> has exactly one pair of equal sides, but these sides do not form a triangle.)

**D6 = {<a, b, c>: a ≥ b + c}**

**D7 = {<a, b, c>: b ≥ a + c}**

**D8 = {<a, b, c>: c ≥ a + b}**

If we wanted to be still more thorough, we could separate the “greater than or equal to” into the two distinct cases; thus, the set D6 would become

$$D6' = \{ \langle a, b, c \rangle : a = b + c \}$$

$$D6'' = \{ \langle a, b, c \rangle : a > b + c \}$$

and similarly for D7 and D8.

## 2.10 Equivalence Class Test Cases for the NextDate Function

The NextDate function illustrates very well the craft of choosing the underlying equivalence relation. Recall that NextDate is a function of three variables: month, day, and year, and these have intervals of valid values defined as follows:

$$M1 = \{ \text{month: } 1 \leq \text{month} \leq 12 \}$$

$$D1 = \{ \text{day: } 1 \leq \text{day} \leq 31 \}$$

$$Y1 = \{ \text{year: } 1812 \leq \text{year} \leq 2012 \}$$

The invalid equivalence classes are

$$M2 = \{ \text{month: month} < 1 \}$$

$$M3 = \{ \text{month: month} > 12 \}$$

$$D2 = \{ \text{day: day} < 1 \}$$

$$D3 = \{ \text{day: day} > 31 \}$$

$$Y2 = \{ \text{year: year} < 1812 \}$$

$$Y3 = \{ \text{year: year} > 2012 \}$$

Because the number of valid classes equals the number of independent variables, only one weak normal equivalence class test case occurs, and it is identical to the strong normal equivalence class test case:

| Case ID  | Month | Day | Year | Expected Output |
|----------|-------|-----|------|-----------------|
| WN1, SN1 | 6     | 15  | 1912 | 6/16/1912       |

Here is the full set of weak robust test cases:

| Case ID | Month | Day | Year | Expected Output                              |
|---------|-------|-----|------|--|
| WR1     | 6     | 15  | 1912 | 6/16/1912                                    |
| WR2     | -1    | 15  | 1912 | Value of month not in the range 1 ... 12     |
| WR3     | 13    | 15  | 1912 | Value of month not in the range 1 ... 12     |
| WR4     | 6     | -1  | 1912 | Value of day not in the range 1 ... 31       |
| WR5     | 6     | 32  | 1912 | Value of day not in the range 1 ... 31       |
| WR6     | 6     | 15  | 1811 | Value of year not in the range 1812 ... 2012 |
| WR7     | 6     | 15  | 2013 | Value of year not in the range 1812 ... 2012 |

with the triangle problem, here is one “corner” of the cube in three-space of the additional strong robust equivalence class test cases:

| <i>Case ID</i> | <i>Month</i> | <i>Day</i> | <i>Year</i> | <i>Expected Output</i>   |
|----------------|--------------|------------|-------------|--|
| SR1            | -1           | 15         | 1912        | Value of month not in the range 1 ... 12   |
| SR2            | 6            | -1         | 1912        | Value of day not in the range 1 ... 31   |
| SR3            | 6            | 15         | 1811        | Value of year not in the range 1812 ... 2012   |
| SR4            | -1           | -1         | 1912        | Value of month not in the range 1 ... 12<br>Value of day not in the range 1 ... 31   |
| SR5            | 6            | -1         | 1811        | Value of day not in the range 1 ... 31<br>Value of year not in the range 1812 ... 2012   |
| SR6            | -1           | 15         | 1811        | Value of month not in the range 1 ... 12<br>Value of year not in the range 1812 ... 2012   |
| SR7            | -1           | -1         | 1811        | Value of month not in the range 1 ... 12<br>Value of day not in the range 1 ... 31<br>Value of year not in the range 1812 ... 2012 |

If we more carefully choose the equivalence relation, the resulting equivalence classes will be more useful. Recall that earlier we said that the gist of the equivalence relation is that elements in a class are “treated the same way.” One way to see the deficiency of the traditional approach is that the “treatment” is at the valid/invalid level. We next reduce the granularity by focusing on more specific treatment.

What must be done to an input date? If it is not the last day of a month, the NextDate function will simply increment the day value. At the end of a month, the next day is 1 and the month is incremented. At the end of a year, both the day and the month are reset to 1, and the year is incremented. Finally, the problem of leap year makes determining the last day of a month interesting. With all this in mind, we might postulate the following equivalence classes:

**M1 = {month: month has 30 days}**

**M2 = {month: month has 31 days}**

**M3 = {month: month is February}**

**D1 = {day:  $1 \leq \text{day} \leq 28$ }**

**D2 = {day: day = 29}**

**D3 = {day: day = 30}**

**D4 = {day: day = 31}**

**Y1 = {year: year = 2000}**

**Y2 = {year: year is a non-century leap year}**

**Y3 = {year: year is a common year}**

By choosing separate classes for 30- and 31-day months, we simplify the question of the last day of the month. By taking February as a separate class, we can give more attention to leap year

questions. We also give special attention to day values: days in D1 are (nearly) always incremented, while days in D4 only have meaning for months in M2. Finally, we have three classes of years, the special case of the year 2000, leap years, and non-leap years. This is not a perfect set of equivalence classes, but its use will reveal many potential errors.

### 2.11 Equivalence Class Test Cases

These classes yield the following weak normal equivalence class test cases. As before, the inputs are mechanically selected from the approximate middle of the corresponding class:

| <i>Case ID</i> | <i>Month</i> | <i>Day</i> | <i>Year</i> | <i>Expected Output</i> |
|----------------|--------------|------------|-------------|------------------------|
| WN1            | 6            | 14         | 2000        | 6/15/2000              |
| WN2            | 7            | 29         | 1996        | 7/30/1996              |
| WN3            | 2            | 30         | 2002        | Invalid input date     |
| WN4            | 6            | 31         | 2000        | Invalid input date     |

Mechanical selection of input values makes no consideration of our domain knowledge, thus the two impossible dates. This will always be a problem with “automatic” test case generation, because all of our domain knowledge is not captured in the choice of equivalence classes. The strong normal equivalence class test cases for the revised classes are as follows:

| <i>Case ID</i> | <i>Month</i> | <i>Day</i> | <i>Year</i> | <i>Expected Output</i> |
|----------------|--------------|------------|-------------|------------------------|
| SN15           | 7            | 14         | 2002        | 7/15/2002              |
| SN16           | 7            | 29         | 2000        | 7/30/2000              |
| SN17           | 7            | 29         | 1996        | 7/30/1996              |
| SN18           | 7            | 29         | 2002        | 7/30/2002              |
| SN19           | 7            | 30         | 2000        | 7/31/2000              |
| SN20           | 7            | 30         | 1996        | 7/31/1996              |
| SN21           | 7            | 30         | 2002        | 7/31/2002              |
| SN22           | 7            | 31         | 2000        | 8/1/2000               |
| SN23           | 7            | 31         | 1996        | 8/1/1996               |
| SN24           | 7            | 31         | 2002        | 8/1/2002               |
| SN25           | 2            | 14         | 2000        | 2/15/2000              |
| SN26           | 2            | 14         | 1996        | 2/15/1996              |
| SN27           | 2            | 14         | 2002        | 2/15/2002              |
| SN28           | 2            | 29         | 2000        | 3/1/2000               |
| SN29           | 2            | 29         | 1996        | 3/1/1996               |
| SN30           | 2            | 29         | 2002        | Invalid input date     |
| SN31           | 2            | 30         | 2000        | Invalid input date     |
| SN32           | 2            | 30         | 1996        | Invalid input date     |
| SN33           | 2            | 30         | 2002        | Invalid input date     |
| SN34           | 2            | 31         | 2000        | Invalid input date     |
| SN35           | 2            | 31         | 1996        | Invalid input date     |
| SN36           | 2            | 31         | 2002        | Invalid input date     |

| <i>Case ID</i> | <i>Month</i> | <i>Day</i> | <i>Year</i> | <i>Expected Output</i> |
|----------------|--------------|------------|-------------|------------------------|
| SN1            | 6            | 14         | 2000        | 6/15/2000              |
| SN2            | 6            | 14         | 1996        | 6/15/1996              |
| SN3            | 6            | 14         | 2002        | 6/15/2002              |
| SN4            | 6            | 29         | 2000        | 6/30/2000              |
| SN5            | 6            | 29         | 1996        | 6/30/1996              |
| SN6            | 6            | 29         | 2002        | 6/30/2002              |
| SN7            | 6            | 30         | 2000        | Invalid input date     |
| SN8            | 6            | 30         | 1996        | Invalid input date     |
| SN9            | 6            | 30         | 2002        | Invalid input date     |
| SN10           | 6            | 31         | 2000        | Invalid input date     |
| SN11           | 6            | 31         | 1996        | Invalid input date     |
| SN12           | 6            | 31         | 2002        | Invalid input date     |
| SN13           | 7            | 14         | 2000        | 7/15/2000              |
| SN14           | 7            | 14         | 1996        | 7/15/1996              |



Moving from weak to strong normal testing raises some of the issues of redundancy that we saw with boundary value testing. The move from weak to strong, whether with normal or robust classes, always makes the presumption of independence, and this is reflected in the cross product of the equivalence classes. Three month classes times four day classes times three year classes results in 36 strong normal equivalence class test cases. Adding two invalid classes for each variable will result in 150 strong robust equivalence class test cases (too many to show here!).

We could also streamline our set of test cases by taking a closer look at the year classes. If we Merge Y1 and Y2, and call the result the set of leap years, our 36 test cases would drop down to 24. This change suppresses special attention to considerations in the year 2000, and it also adds some Complexity to the determination of which years are leap years. Balance this against how much might be learned from the present test cases.

## 2.12 Equivalence Class Test Cases for the Commission Problem

The input domain of the commission problem is “naturally” partitioned by the limits on locks, stocks, and barrels. These equivalence classes are exactly those that would also be identified by traditional equivalence class testing. The first class is the valid input; the other two are invalid. The input domain equivalence classes lead to very unsatisfactory sets of test cases. Equivalence classes defined on the output range of the commission function will be an improvement.

The valid classes of the input variables are

**L1 = {locks:  $1 \leq \text{locks} \leq 70$ }**

**L2 = {locks = -1} (occurs if locks = -1 is used to control input iteration)**

**S1 = {stocks:  $1 \leq \text{stocks} \leq 80$ }**

**B1 = {barrels:  $1 \leq \text{barrels} \leq 90$ }**

The corresponding invalid classes of the input variables are

**L3 = {locks: locks = 0 OR locks < -1}**

**L4 = {locks: locks > 70}**

**S2 = {stocks: stocks < 1}**

**S3 = {stocks: stocks > 80}**

**B2 = {barrels: barrels < 1}**

**B3 = {barrels: barrels > 90}**

One problem occurs, however. The variable “locks” is also used as a sentinel to indicate no more telegrams. When a value of -1 is given for locks, the while loop terminates, and the values of totalLocks, totalStocks, and totalBarrels are used to compute sales, and then commission.

Except for the names of the variables and the interval endpoint values, this is identical to our first version of the NextDate function. Therefore, we will have exactly one weak normal equivalence class test case—and again, it is identical to the strong normal equivalence class test case. Note that the case for locks = -1 just terminates the iteration. We will have eight weak robust test cases.

| <i>Case ID</i> | <i>Locks</i> | <i>Stocks</i> | <i>Barrels</i> | <i>Expected Output</i>                     |
|----------------|--------------|---------------|----------------|--|
| WR1            | 10           | 10            | 10             | \$100                                      |
| WR2            | -1           | 40            | 45             | Program terminates                         |
| WR3            | -2           | 40            | 45             | Value of locks not in the range 1 ... 70   |
| WR4            | 71           | 40            | 45             | Value of locks not in the range 1 ... 70   |
| WR5            | 35           | -1            | 45             | Value of stocks not in the range 1 ... 80  |
| WR6            | 35           | 81            | 45             | Value of stocks not in the range 1 ... 80  |
| WR7            | 35           | 40            | -1             | Value of barrels not in the range 1 ... 90 |
| WR8            | 35           | 40            | 91             | Value of barrels not in the range 1 ... 90 |

Here is one “corner” of the cube in 3-space of the additional strong robust equivalence class test cases:

| <i>Case ID</i> | <i>Locks</i> | <i>Stocks</i> | <i>Barrels</i> | <i>Expected Output</i>  |
|----------------|--------------|---------------|----------------|---|
| SR1            | -2           | 40            | 45             | Value of locks not in the range 1 ... 70  |
| SR2            | 35           | -1            | 45             | Value of stocks not in the range 1 ... 80   |
| SR3            | 35           | 40            | -2             | Value of barrels not in the range 1 ... 90  |
| SR4            | -2           | -1            | 45             | Value of locks not in the range 1 ... 70<br>Value of stocks not in the range 1 ... 80   |
| SR5            | -2           | 40            | -1             | Value of locks not in the range 1 ... 70<br>Value of barrels not in the range 1 ... 90  |
| SR6            | 35           | -1            | -1             | Value of stocks not in the range 1 ... 80<br>Value of barrels not in the range 1 ... 90   |
| SR7            | -2           | -1            | -1             | Value of locks not in the range 1 ... 70<br>Value of stocks not in the range 1 ... 80<br>Value of barrels not in the range 1 ... 90 |

Notice that, of strong test cases—whether normal or robust—only one is a legitimate input. If we were really worried about error cases, this might be a good set of test cases. It can hardly give us a sense of confidence about the calculation portion of the problem, however. We can get some help by considering equivalence classes defined on the output range. Recall that sales is a function of the number of locks, stocks, and barrels sold:

$$\text{Sales} = 45 \times \text{locks} + 30 \times \text{stocks} + 25 \times \text{barrels}$$

We could define equivalence classes of three variables by commission ranges:

**S1 = {<locks, stocks, barrels>: sales  $\leq$  1000}**

**S2 = {<locks, stocks, barrels>: 1000 < sales  $\leq$  1800}**

**S3 = {<locks, stocks, barrels>: sales > 1800}**

Figure 2.6 helps us get a better feel for the input space. Elements of S1 are points with integer coordinates in the pyramid near the origin. Elements of S2 are points in the “triangular slice” between the pyramid and the rest of the input space. Finally, elements of S3 are all those points in the rectangular volume that are not in S1 or in S2. All the error cases found by the strong equivalence classes of the input domain are outside of the rectangular space shown in Figure 2.6.

As was the case with the triangle problem, the fact that our input is a triplet means that we no longer take test cases from a Cartesian product.

| <i>Test Case</i> | <i>Locks</i> | <i>Stocks</i> | <i>Barrels</i> | <i>Sales</i> | <i>Commission</i> |
|------------------|--------------|---------------|----------------|--------------|-------------------|
| OR1              | 5            | 5             | 5              | 500          | 50                |
| OR2              | 15           | 15            | 15             | 1500         | 175               |
| OR3              | 25           | 25            | 25             | 2500         | 360               |

These test cases give us some sense that we are exercising important parts of the problem. Together with the weak robust test cases, we would have a pretty good test of the commission problem. We might want to add some boundary checking, just to make sure the transitions at sales of \$1000 and \$1800 are correct. This is not particularly easy because we can only choose values of locks, stocks, and barrels. It happens that the constants in this example are contrived so that there are “nice” triplets.

### 2.13 Guidelines and Observations

Now that we have gone through three examples, we conclude with some observations about, and guidelines for, equivalence class testing.

1. Obviously, the weak forms of equivalence class testing (normal or robust) are not as comprehensive as the corresponding strong forms.
2. If the implementation language is strongly typed (and invalid values cause run-time errors), It makes no sense to use the robust forms.
3. If error conditions are a high priority, the robust forms are appropriate.
4. Equivalence class testing is appropriate when input data is defined in terms of intervals and sets of discrete values. This is certainly the case when system malfunctions can occur for out-of-limit variable values.
5. Equivalence class testing is strengthened by a hybrid approach with boundary value testing. (We can “reuse” the effort made in defining the equivalence classes.)



6. Equivalence class testing is indicated when the program function is complex. In such cases, the complexity of the function can help identify useful equivalence classes, as in the NextDate function.
7. Strong equivalence class testing makes a presumption that the variables are independent, and the corresponding multiplication of test cases raises issues of redundancy. If any dependencies occur, they will often generate “error” test cases, as they did in the NextDate function.
8. Several tries may be needed before the “right” equivalence relation is discovered, as we saw in the NextDate example. In other cases, there is an “obvious” or “natural” equivalence relation. When in doubt, the best bet is to try to second-guess aspects of any reasonable implementation. This is sometimes known as the “competent programmer hypothesis.”
9. The difference between the strong and weak forms of equivalence class testing is helpful in the distinction between progression and regression testing.

## Decision Table–Based Testing

### 2.14 Decision Tables

Decision tables have been used to represent and analyze complex logical relationships since the early 1960s. They are ideal for describing situations in which a number of combinations of actions are taken under varying sets of conditions. Some of the basic decision table terms are illustrated in Table 2.6.

A decision table has four portions: the part to the left of the bold vertical line is the stub portion; to the right is the entry portion. The part above the bold horizontal line is the condition portion, and below is the action portion. Thus, we can refer to the condition stub, the condition entries, the action stub, and the action entries. A column in the entry portion is a rule. Rules indicate which actions, if any, are taken for the circumstances indicated in the condition portion of the rule. In the decision table in Table 2.6, when conditions  $c_1$ ,  $c_2$ , and  $c_3$  are all true, actions  $a_1$  and  $a_2$  occur. When  $c_1$  and  $c_2$  are both true and  $c_3$  is false, then actions  $a_1$  and  $a_3$  occur. The entry for  $c_3$  in the rule where  $c_1$  is true and  $c_2$  is false is called a “don’t care” entry. The don’t care entry has two major interpretations: the condition is irrelevant, or the condition does not apply. Sometimes people will enter the “n/a” symbol for this latter interpretation.

When we have binary conditions (true/false, yes/no, 0/1), the condition portion of a decision table is a truth table (from propositional logic) that has been rotated  $90^\circ$ . This structure guarantees that we consider every possible combination of condition values. When we use decision tables for test case identification, this completeness property of a decision table guarantees a form of complete testing. Decision tables in which all the conditions are binary are called Limited Entry Decision Tables (LETDs). If conditions are allowed to have several values, the resulting tables are called Extended Entry Decision Tables (EEDTs). We will see examples of both types for the NextDate problem. Decision tables are deliberately declarative (as opposed to imperative); no particular order is implied by the conditions, and selected actions do not occur in any particular order.

**Table 2.6 Portions of a Decision Table**

| <i>Stub</i> | <i>Rule 1</i> | <i>Rule 2</i> | <i>Rules 3, 4</i> | <i>Rule 5</i> | <i>Rule 6</i> | <i>Rules 7, 8</i> |
|-------------|---------------|---------------|-------------------|---------------|---------------|-------------------|
| c1          | T             | T             | T                 | F             | F             | F                 |
| c2          | T             | T             | F                 | T             | T             | F                 |
| c3          | T             | F             | —                 | T             | F             | —                 |
| a1          | X             | X             |                   | X             |               |                   |
| a2          | X             |               |                   |               | X             |                   |
| a3          |               | X             |                   | X             |               |                   |
| a4          |               |               | X                 |               |               | X                 |

### 2.15 Decision Table Techniques

To identify test cases with decision tables, we interpret conditions as inputs and actions as outputs.

Sometimes conditions end up referring to equivalence classes of inputs, and actions refer to major functional processing portions of the item tested. The rules are then interpreted as test cases.

Because the decision table can mechanically be forced to be complete, we have some assurance that we will have a comprehensive set of test cases. Several techniques that produce decision tables are more useful to testers. One helpful style is to add an action to show when a rule is logically impossible.

In the decision table in Table 2.7, we see examples of don't care entries and impossible rule usage. If the integers a, b, and c do not constitute a triangle, we do not even care about possible

**Table 2.7 Decision Table for Triangle Problem**

|                              |   |   |   |   |   |   |   |   |   |
|------------------------------|---|---|---|---|---|---|---|---|---|
| c1: a, b, c form a triangle? | F | T | T | T | T | T | T | T | T |
| c2: a = b?                   | — | T | T | T | T | F | F | F | F |
| c3: a = c?                   | — | T | T | F | F | T | T | F | F |
| c4: b = c?                   | — | T | F | T | F | T | F | T | F |
| a1: Not a triangle           | X |   |   |   |   |   |   |   |   |
| a2: Scalene                  |   |   |   |   |   |   |   |   | X |
| a3: Isosceles                |   |   |   |   | X |   | X | X |   |
| a4: Equilateral              |   | X |   |   |   |   |   |   |   |
| a5: Impossible               |   |   | X | X |   | X |   |   |   |

equalities, as indicated in the first rule. In rules 3, 4, and 6, if two pairs of integers are equal, by transitivity, the third pair must be equal; thus, the negative entry makes these rules impossible.

The decision table in Table 2.8 illustrates another consideration: the choice of conditions can greatly expand the size of a decision table. Here, we have expanded the old condition (c1: a, b, c form a triangle?) to a more detailed view of the three inequalities of the triangle property. If any one of these fails, the three integers do not constitute sides of a triangle.

We could expand this still further because there are two ways an inequality could fail: one side could equal the sum of the other two, or it could be strictly greater.

When conditions refer to equivalence classes, decision tables have a characteristic appearance.

Conditions in the decision table in Table 2.9 are from the NextDate problem; they refer to the mutually exclusive possibilities for the month variable. Because a month is in exactly one equivalence class, we cannot ever have a rule in which two entries are true. The don't care entries (—)

really mean “must be false.” Some decision table aficionados use the notation  $F!$  to make this point.

Use of don't care entries has a subtle effect on the way in which complete decision tables are recognized. For a limited entry decision table with  $n$  conditions, there must be  $2n$  independent

**Table 2.8 Refined Decision Table for Triangle Problem**

|                    |   |   |   |   |   |   |   |   |   |   |   |
|--------------------|---|---|---|---|---|---|---|---|---|---|---|
| c1: $a < b + c?$   | F | T | T | T | T | T | T | T | T | T | T |
| c2: $b < a + c?$   | — | F | T | T | T | T | T | T | T | T | T |
| c3: $c < a + b?$   | — | — | F | T | T | T | T | T | T | T | T |
| c4: $a = b?$       | — | — | — | T | T | T | T | F | F | F | F |
| c5: $a = c?$       | — | — | — | T | T | F | F | T | T | F | F |
| c6: $b = c?$       | — | — | — | T | F | T | F | T | F | T | F |
| a1: Not a triangle | X | X | X |   |   |   |   |   |   |   |   |
| a2: Scalene        |   |   |   |   |   |   |   |   |   |   | X |
| a3: Isosceles      |   |   |   |   |   |   | X |   | X | X |   |
| a4: Equilateral    |   |   |   | X |   |   |   |   |   |   |   |
| a5: Impossible     |   |   |   |   | X | X |   | X |   |   |   |

**Table 2.9 Decision Table with Mutually Exclusive Conditions**

| Conditions       | R1 | R2 | R3 |
|------------------|----|----|----|
| c1: Month in M1? | T  | —  | —  |
| c2: Month in M2? | —  | T  | —  |
| c3: Month in M3? | —  | —  | T  |
| a1               |    |    |    |
| a2               |    |    |    |
| a3               |    |    |    |

rules. When don't care entries really indicate that the condition is irrelevant, we can develop a rule count as follows: rules in which no don't care entries occur count as one rule, and each don't care entry in a rule doubles the count of that rule. The rule counts for the decision table in Table 7.3 are shown in Table 2.10. Notice that the sum of the rule counts is 64 (as it should be).

If we applied this simplistic algorithm to the decision table in Table 7.4, we get the rule counts shown in Table 2.11. We should only have eight rules, so we clearly have a problem. To see where the problem lies, we expand each of the three rules, replacing the “—” entries with the T and F possibilities, as shown in Table 2.12. Notice that we have three rules in which all entries are T: rules 1.1, 2.1, and 3.1. We also have two rules with T, T, F entries: rules 1.2 and 2.2. Similarly, rules 1.3 and 3.2 are identical; so are rules 2.3 and 3.3. If we delete the repetitions, we end up with seven rules; the missing rule is the one in which all conditions are false. The result of this process is shown in Table 2.13. The impossible rules are also shown.

**Table 2.10 Decision Table for Table 2.8 with Rule Counts**

|                    |    |    |   |   |   |   |   |   |   |   |   |
|--------------------|----|----|---|---|---|---|---|---|---|---|---|
| c1: $a < b + c$ ?  | F  | T  | T | T | T | T | T | T | T | T | T |
| c2: $b < a + c$ ?  | —  | F  | T | T | T | T | T | T | T | T | T |
| c3: $c < a + b$ ?  | —  | —  | F | T | T | T | T | T | T | T | T |
| c4: $a = b$ ?      | —  | —  | — | T | T | T | T | F | F | F | F |
| c5: $a = c$ ?      | —  | —  | — | T | T | F | F | T | T | F | F |
| c6: $b = c$ ?      | —  | —  | — | T | F | T | F | T | F | T | F |
| Rule count         | 32 | 16 | 8 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| a1: Not a triangle | X  | X  | X |   |   |   |   |   |   |   |   |
| a2: Scalene        |    |    |   |   |   |   |   |   |   |   | X |
| a3: Isosceles      |    |    |   |   |   |   | X |   | X | X |   |
| a4: Equilateral    |    |    |   | X |   |   |   |   |   |   |   |
| a5: Impossible     |    |    |   |   | X | X |   | X |   |   |   |

**Table 2.11 Rule Counts for a Decision Table with Mutually Exclusive Conditions**

| <i>Conditions</i> | <i>R1</i> | <i>R2</i> | <i>R3</i> |
|-------------------|-----------|-----------|-----------|
| c1: Month in M1   | T         | —         | —         |
| c2: Month in M2   | —         | T         | —         |
| c3: Month in M3   | —         | —         | T         |
| Rule count        | 4         | 4         | 4         |
| a1                |           |           |           |

**Table 2.12 Impossible Rules in Table 2.12**

| <i>Conditions</i> | <i>1.1</i> | <i>1.2</i> | <i>1.3</i> | <i>1.4</i> | <i>2.1</i> | <i>2.2</i> | <i>2.3</i> | <i>2.4</i> | <i>3.1</i> | <i>3.2</i> | <i>3.3</i> | <i>3.4</i> |
|-------------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|------------|
| c1: Month in M1   | T          | T          | T          | T          | T          | T          | F          | F          | T          | T          | F          | F          |
| c2: Month in M2   | T          | T          | F          | F          | T          | T          | T          | T          | T          | F          | T          | F          |
| c3: Month in M3   | T          | F          | T          | F          | T          | F          | T          | F          | T          | T          | T          | T          |
| Rule count        | 1          | 1          | 1          | 1          | 1          | 1          | 1          | 1          | 1          | 1          | 1          | 1          |
| a1: Impossible    | X          | X          | X          | —          | X          | X          | X          | —          | X          | X          | —          |            |

**Table 2.13 Mutually Exclusive Conditions with Impossible Rules**

|                 | <i>1.1</i> | <i>1.2</i> | <i>1.3</i> | <i>1.4</i> | <i>2.3</i> | <i>2.4</i> | <i>3.4</i> |   |
|-----------------|------------|------------|------------|------------|------------|------------|------------|---|
| c1: Month in M1 | T          | T          | T          | T          | F          | F          | F          | F |
| c2: Month in M2 | T          | T          | F          | F          | T          | T          | F          | F |
| c3: Month in M3 | T          | F          | T          | F          | T          | F          | T          | F |
| Rule count      | 1          | 1          | 1          | 1          | 1          | 1          | 1          | 1 |
| a1: Impossible  | X          | X          | X          |            | X          |            |            | X |

**Table 2.14 A Redundant Decision Table**

| <i>Conditions</i> | <i>1–4</i> | <i>5</i> | <i>6</i> | <i>7</i> | <i>8</i> | <i>9</i> |
|-------------------|------------|----------|----------|----------|----------|----------|
| c1                | T          | F        | F        | F        | F        | T        |
| c2                | —          | T        | T        | F        | F        | F        |
| c3                | —          | T        | F        | T        | F        | F        |
| a1                | X          | X        | X        | —        | —        | X        |
| a2                | —          | X        | X        | X        | —        | —        |
| a3                | X          | —        | X        | X        | X        | X        |

The ability to recognize (and develop) complete decision tables puts us in a powerful position with respect to redundancy and inconsistency. The decision table in Table 2.14 is redundant—three conditions and nine rules exist. (Rule 9 is identical to rule 4.) Notice that the action entries in rule 9 are identical to those in rules 1–4. As long as the actions in a redundant rule are identical to the corresponding part of the decision table, we do not have much of a problem. If the action entries are different, as in Table 2.15, we have a bigger problem.

If the decision table in Table 2.15 were to process a transaction in which  $c_1$  is true and both  $c_2$  and  $c_3$  are false, both rules 4 and 9 apply. We can make two observations:

1. Rules 4 and 9 are inconsistent.
2. The decision table is nondeterministic.

**Table 2.15 A n Inconsistent Decision Table**

| <i>Conditions</i> | <i>1–4</i> | <i>5</i> | <i>6</i> | <i>7</i> | <i>8</i> | <i>9</i> |
|-------------------|------------|----------|----------|----------|----------|----------|
| $c_1$             | T          | F        | F        | F        | F        | T        |
| $c_2$             | —          | T        | T        | F        | F        | F        |
| $c_3$             | —          | T        | F        | T        | F        | F        |
| $a_1$             | X          | X        | X        | —        | —        | —        |
| $a_2$             | —          | X        | X        | X        | —        | X        |
| $a_3$             | X          | —        | X        | X        | X        | —        |

Rules 4 and 9 are inconsistent because the action sets are different. The whole table is nondeterministic because there is no way to decide whether to apply rule 4 or rule 9. The bottom line for testers is that care should be taken when don't care entries are used in a decision table.

## 2.16 Test Cases for the Triangle Problem

Using the decision table in Table 2.16, we obtain 11 functional test cases: three impossible cases, three ways to fail the triangle property, one way to get an equilateral triangle, one way to get a scalene triangle, and three ways to get an isosceles triangle. We still need to provide

**Table 2.16 Test Cases from Table 2.16**

| <i>Case ID</i> | <i>a</i> | <i>b</i> | <i>c</i> | <i>Expected Output</i> |
|----------------|----------|----------|----------|------------------------|
| DT1            | 4        | 1        | 2        | Not a triangle         |
| DT2            | 1        | 4        | 2        | Not a triangle         |
| DT3            | 1        | 2        | 4        | Not a triangle         |
| DT4            | 5        | 5        | 5        | Equilateral            |
| DT5            | ?        | ?        | ?        | Impossible             |
| DT6            | ?        | ?        | ?        | Impossible             |
| DT7            | 2        | 2        | 3        | Isosceles              |
| DT8            | ?        | ?        | ?        | Impossible             |
| DT9            | 2        | 3        | 2        | Isosceles              |
| DT10           | 3        | 2        | 2        | Isosceles              |
| DT11           | 3        | 4        | 5        | Scalene                |

actual values for the variables in the conditions, but we cannot do this for the impossible rules. If we extended the decision table to show both ways to fail an inequality, we would pick up three more test cases (where one side is exactly the sum of the other two). Some judgment is required in this because of the exponential growth of rules. In this case, we would end up with many more don't care entries and more impossible rules.

### 2.17 Test Cases for the NextDate Function

The NextDate function was chosen because it illustrates the problem of dependencies in the input domain. This makes it a perfect example for decision table-based testing, because decision tables can highlight such dependencies. Recall that, in Chapter 6, we identified equivalence classes in the input domain of the NextDate function. One of the limitations we found in Chapter 6 was that indiscriminate selection of input values from the equivalence classes resulted in “strange” test cases, such as finding the next date to June 31, 1812. The problem stems from the presumption that the variables are independent. If they are, a Cartesian product of the classes makes sense. When logical dependencies exist among variables in the input domain, these dependencies are lost (suppressed is better) in a Cartesian product. The decision table format lets us emphasize such dependencies using the notion of the “impossible” action to denote impossible combinations of conditions (which are actually impossible rules). In this section, we will make three tries at a decision table formulation of the NextDate function.

#### 2.17.1 First Try

Identifying appropriate conditions and actions presents an opportunity for craftsmanship. Suppose we start with a set of equivalence classes close to the one we used in Chapter 6.

M1 = {month: month has 30 days}

M2 = {month: month has 31 days}

M3 = {month: month is February}

D1 = {day:  $1 \leq \text{day} \leq 28$ }

D2 = {day: day = 29}

D3 = {day: day = 30}

D4 = {day: day = 31}

Y1 = {year: year is a leap year}

Y2 = {year: year is not a leap year}

If we wish to highlight impossible combinations, we could make a limited entry decision table with the following conditions and actions. (Note that the equivalence classes for the year variable collapse into one condition in Table 2.17.)

This decision table will have 256 rules, many of which will be impossible. If we wanted to show why these rules were impossible, we might revise our actions to the following:

a1: Day invalid for this month

a2: Cannot happen in a non-leap year

a3: Compute the next date

**Table 2.17 First Try Decision Table with 256 Rules**

| <i>Conditions</i> |   |   |   |  |  |  |  |  |  |  |  |
|-------------------|---|---|---|--|--|--|--|--|--|--|--|
| c1: Month in M1?  | T |   |   |  |  |  |  |  |  |  |  |
| c2: Month in M2?  |   | T |   |  |  |  |  |  |  |  |  |
| c3: Month in M3?  |   |   | T |  |  |  |  |  |  |  |  |
| c4: Day in D1?    |   |   |   |  |  |  |  |  |  |  |  |
| c5: Day in D2?    |   |   |   |  |  |  |  |  |  |  |  |
| c6: Day in D3?    |   |   |   |  |  |  |  |  |  |  |  |
| c7: Day in D4?    |   |   |   |  |  |  |  |  |  |  |  |
| c8: Year in Y1?   |   |   |   |  |  |  |  |  |  |  |  |
| a1: Impossible    |   |   |   |  |  |  |  |  |  |  |  |
| a2: Next date     |   |   |   |  |  |  |  |  |  |  |  |

### 2.17.2 Second Try

If we focus on the leap year aspect of the NextDate function, we could use the set of equivalence classes as they were in Chapter 6. These classes have a Cartesian product that contains 36 triples, with several that are impossible.

To illustrate another decision table technique, this time we will develop an extended entry decision table, and we will take a closer look at the action stub. In making an extended entry decision table, we must ensure that the equivalence classes form a true partition of the input domain.

(Recall from Chapter 3 that a partition is a set of disjoint subsets where the union is the entire set.) If there were any “overlaps” among the rule entries, we would have a redundant case in which more than one rule could be satisfied. Here, Y2 is the set of years between 1812 and 2012, evenly divisible by four excluding the year 2000.

M1 = {month: month has 30 days}

M2 = {month: month has 31 days}

M3 = {month: month is February}

D1 = {day:  $1 \leq \text{day} \leq 28$ }

D2 = {day: day = 29}

D3 = {day: day = 30}

D4 = {day: day = 31}

Y1 = {year: year = 2000}

Y2 = {year: year is a non-century leap year}

Y3 = {year: year is a common year}



In a sense, we could argue that we have a “gray box” technique, because we take a closer look at the NextDate problem statement. To produce the next date of a given date, only five possible actions are needed: incrementing and resetting the day and month, and incrementing the year. (We will not let time go backward by resetting the year.) To follow the metaphor, we still cannot see inside the implementation box—the implementation could be a table look-up.

These conditions would result in a decision table with 36 rules that correspond to the Cartesian product of the equivalence classes. Combining rules with don't care entries yields the decision table in Table 2.18, which has 16 rules. We still have the problem with logically impossible rules, but this formulation helps us identify the expected outputs of a test case. If you complete the action entries in this table, you will find some cumbersome problems with December (in rule 8) and other problems with Feb. 28 in rules 9, 11, and 12. We fix these next.

**Table 2.18 Second Try Decision Table with 36 Rules**

|                     | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  |
|---------------------|----|----|----|----|----|----|----|----|
| c1: Month in        | M1 | M1 | M1 | M1 | M2 | M2 | M2 | M2 |
| c2: Day in          | D1 | D2 | D3 | D4 | D1 | D2 | D3 | D4 |
| c3: Year in         | —  | —  | —  | —  | —  | —  | —  | —  |
| Rule count          | 3  | 3  | 3  | 3  | 3  | 3  | 3  | 3  |
| <b>Actions</b>      |    |    |    |    |    |    |    |    |
| a1: Impossible      |    |    |    | X  |    |    |    |    |
| a2: Increment day   | X  | X  |    |    | X  | X  | X  |    |
| a3: Reset day       |    |    | X  |    |    |    |    | X  |
| a4: Increment month |    |    | X  |    |    |    |    | ?  |
| a5: Reset month     |    |    |    |    |    |    |    | ?  |
| a6: Increment year  |    |    |    |    |    |    |    | ?  |
|                     | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| c1: Month in        | M3 | M3 | M3 | M3 | M3 | M3 | M3 | M3 |
| c2: Day in          | D1 | D1 | D1 | D2 | D2 | D2 | D3 | D4 |
| c3: Year in         | Y1 | Y2 | Y3 | Y1 | Y2 | Y3 | —  | —  |
| Rule count          | 1  | 1  | 1  | 1  | 1  | 1  | 3  | 3  |
| <b>Actions</b>      |    |    |    |    |    |    |    |    |
| a1: Impossible      |    |    |    |    |    | X  | X  | X  |
| a2: Increment day   | X  | X  | ?  |    |    |    |    |    |
| a3: Reset day       |    |    | ?  | X  | X  |    |    |    |
| a4: Increment month | X  |    | X  | X  | X  |    |    |    |
| a5: Reset month     |    |    |    |    |    |    |    |    |
| a6: Increment year  |    |    |    |    |    |    |    |    |

### 7.4.3 Third Try

We can clear up the end-of-year considerations with a third set of equivalence classes. This time, we are very specific about days and months, and we revert to the simpler leap year or non-leap year condition of the first try—so the year 2000 gets no special attention. (We could do a fourth try, showing year equivalence classes as in the second try, but by now you get the point.)

$M1 = \{\text{month: month has 30 days}\}$   
 $M2 = \{\text{month: month has 31 days except December}\}$   
 $M3 = \{\text{month: month is December}\}$   
 $M4 = \{\text{month: month is February}\}$   
 $D1 = \{\text{day: } 1 \leq \text{day} \leq 27\}$   
 $D2 = \{\text{day: day} = 28\}$   
 $D3 = \{\text{day: day} = 29\}$   
 $D4 = \{\text{day: day} = 30\}$   
 $D5 = \{\text{day: day} = 31\}$   
 $Y1 = \{\text{year: year is a leap year}\}$   
 $Y2 = \{\text{year: year is a common year}\}$

The Cartesian product of these contains 40 elements. The result of combining rules with don't care entries is given in Table 2.19; it has 22 rules, compared with the 36 of the second try. Recall from Chapter 1 the question of whether a large set of test cases is necessarily better than a smaller set. Here, we have a 22-rule decision table that gives a clearer picture of the NextDate function than does the 36-rule decision table. The first five rules deal with 30-day months; notice that the leap year considerations are irrelevant. The next two sets of rules (6–15) deal with 31-day months, where rules 6–10 deal with months other than December and rules 11–15 deal with December. No impossible rules are listed in this portion of the decision table, although there is some redundancy that an efficient tester might question. Eight of the 10 rules simply increment the day. Would we really require eight separate test cases for this subfunction? Probably not; but note the insights we can get from the decision table. Finally, the last seven rules focus on February in common and leap years.

The decision table in Table 2.19 is the basis for the source code for the NextDate function in Chapter 2. As an aside, this example shows how good testing can improve programming. All the decision table analysis could have been done during the detailed design of the NextDate function.

We can use the algebra of decision tables to further simplify these 22 test cases. If the action sets of two rules in a limited entry decision table are identical, there must be at least one condition that allows two rules to be combined with a don't care entry. This is the decision table equivalent of the “treated the same” guideline that we used to identify equivalence classes. In a sense, we are identifying equivalence classes of rules. For example, rules 1, 2, and 3 involve day classes D1, D2, and D3 for 30-day months. These can be combined similarly for day classes D1, D2, D3, and D4 in the 31-day month rules, and D4 and D5 for February. The result is in Table 2.28.

The corresponding test cases are shown in Table 2.21.

**Table 2.19 Decision Table for NextDate Function**

|                     | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 |    |    |
|---------------------|----|----|----|----|----|----|----|----|----|----|----|----|
| c1: Month in        | M1 | M1 | M1 | M1 | M1 | M2 | M2 | M2 | M2 | M2 |    |    |
| c2: Day in          | D1 | D2 | D3 | D4 | D5 | D1 | D2 | D3 | D4 | D5 |    |    |
| c3: Year in         | —  | —  | —  | —  | —  | —  | —  | —  | —  | —  |    |    |
| <b>Actions</b>      |    |    |    |    |    |    |    |    |    |    |    |    |
| a1: Impossible      |    |    |    |    | X  |    |    |    |    |    |    |    |
| a2: Increment day   | X  | X  | X  |    |    | X  | X  | X  | X  |    |    |    |
| a3: Reset day       |    |    |    | X  |    |    |    |    |    | X  |    |    |
| a4: Increment month |    |    |    | X  |    |    |    |    |    | X  |    |    |
| a5: Reset month     |    |    |    |    |    |    |    |    |    |    |    |    |
| a6: Increment year  |    |    |    |    |    |    |    |    |    |    |    |    |
|                     | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 |
| c1: Month in        | M3 | M3 | M3 | M3 | M3 | M4 | M4 | M4 | M4 | M4 | M4 | M4 |
| c2: Day in          | D1 | D2 | D3 | D4 | D5 | D1 | D2 | D2 | D3 | D3 | D4 | D5 |
| c3: Year in         | —  | —  | —  | —  | —  | —  | Y1 | Y2 | Y1 | Y2 | —  | —  |
| <b>Actions</b>      |    |    |    |    |    |    |    |    |    |    |    |    |
| a1: Impossible      |    |    |    |    |    |    |    |    |    | X  | X  | X  |
| a2: Increment day   | X  | X  | X  | X  |    | X  | X  |    |    |    |    |    |
| a3: Reset day       |    |    |    |    | X  |    |    | X  | X  |    |    |    |
| a4: Increment month |    |    |    |    |    |    |    | X  | X  |    |    |    |
| a5: Reset month     |    |    |    |    | X  |    |    |    |    |    |    |    |
| a6: Increment year  |    |    |    |    | X  |    |    |    |    |    |    |    |

**Table 2.20 Reduced Decision Table for NextDate Function**

|                     | 1-3            | 4  | 5  | 6-9            | 10 |    |    |        |
|---------------------|----------------|----|----|----------------|----|----|----|--------|
| c1: Month in        | M1             | M1 | M1 | M2             | M2 |    |    |        |
| c2: Day in          | D1, D2, D3     | D4 | D5 | D1, D2, D3, D4 | D5 |    |    |        |
| c3: Year in         | —              | —  | —  | —              | —  |    |    |        |
| <b>Actions</b>      |                |    |    |                |    |    |    |        |
| a1: Impossible      |                |    | X  |                |    |    |    |        |
| a2: Increment day   | X              |    |    | X              |    |    |    |        |
| a3: Reset day       |                | X  |    |                | X  |    |    |        |
| a4: Increment month |                | X  |    |                | X  |    |    |        |
| a5: Reset month     |                |    |    |                |    |    |    |        |
| a6: Increment year  |                |    |    |                |    |    |    |        |
|                     | 11-14          | 15 | 16 | 17             | 18 | 19 | 20 | 21, 22 |
| c1: Month in        | M3             | M3 | M4 | M4             | M4 | M4 | M4 | M4     |
| c2: Day in          | D1, D2, D3, D4 | D5 | D1 | D2             | D2 | D3 | D3 | D4, D5 |
| c3: Year in         | —              | —  | —  | Y1             | Y2 | Y1 | Y2 | —      |
| <b>Actions</b>      |                |    |    |                |    |    |    |        |
| a1: Impossible      |                |    |    |                |    |    | X  | X      |
| a2: Increment day   | X              |    | X  | X              |    |    |    |        |
| a3: Reset day       |                | X  |    |                | X  | X  |    |        |
| a4: Increment month |                |    |    |                | X  | X  |    |        |
| a5: Reset month     |                | X  |    |                |    |    |    |        |
| a6: Increment year  |                | X  |    |                |    |    |    |        |

### 2.18 Test Cases for the Commission Problem

The commission problem is not well served by a decision table analysis. This is not surprising because very little decisional logic is used in the problem. Because the variables in the equivalence classes are truly independent, no impossible rules will occur in a decision table in which conditions correspond to the equivalence classes. Thus, we will have the same test cases as we did for equivalence class testing.

### 2.19 Cause-and-Effect Graphing

In the early years of computing, the software community borrowed many ideas from the hardware community. In some cases this worked well, but in others, the problems of software just did not fit well with established hardware techniques. Cause-and-effect graphing is a good example of this. The base hardware concept was the practice of describing circuits composed of discrete components with AND, OR, and NOT gates. There was usually an input side of a circuit diagram, and

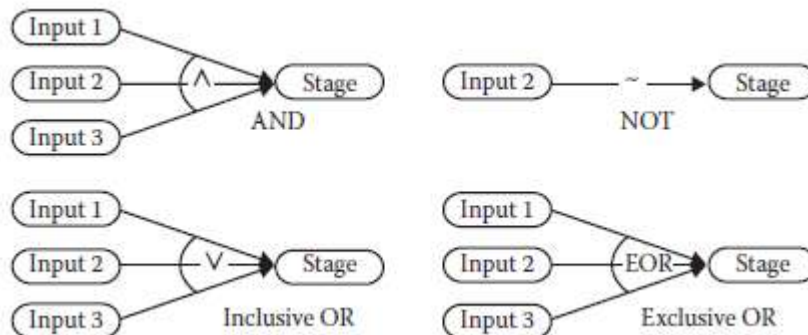
**Table 2.21 Decision Table Test Cases for NextDate**

| <i>Case ID</i> | <i>Month</i> | <i>Day</i> | <i>Year</i> | <i>Expected Output</i> |
|----------------|--------------|------------|-------------|------------------------|
| 1–3            | 4            | 15         | 2001        | 4/16/2001              |
| 4              | 4            | 30         | 2001        | 5/1/2001               |
| 5              | 4            | 31         | 2001        | Invalid input date     |
| 6–9            | 1            | 15         | 2001        | 1/16/2001              |
| 10             | 1            | 31         | 2001        | 2/1/2001               |
| 11–14          | 12           | 15         | 2001        | 12/16/2001             |
| 15             | 12           | 31         | 2001        | 1/1/2002               |
| 16             | 2            | 15         | 2001        | 2/16/2001              |
| 17             | 2            | 28         | 2004        | 2/29/2004              |
| 18             | 2            | 28         | 2001        | 3/1/2001               |
| 19             | 2            | 29         | 2004        | 3/1/2004               |
| 20             | 2            | 29         | 2001        | Invalid input date     |
| 21, 22         | 2            | 30         | 2001        | Invalid input date     |

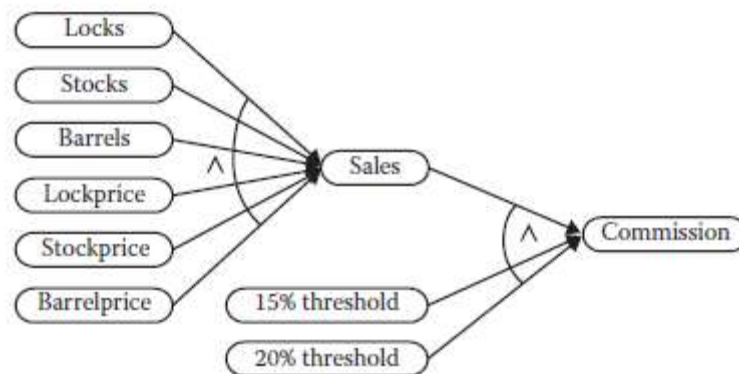
the flow of inputs through the various components could be generally traced from left to right. With this, the effects of hardware faults such as stuck-at-one/zero could be traced to the output side. This greatly facilitated circuit testing.

Cause-and-effect graphs attempt to follow this pattern, by showing unit inputs on the left side of a drawing, and using AND, OR, and NOT “gates” to express the flow of data across stages of a unit. Figure 2.12 shows the basic cause-and-effect graph structures. The basic structures can be augmented by less used operations: Identity, Masks, Requires, and Only One.

The most that can be learned from a cause-and-effect graph is that, if there is a problem at an output, the path(s) back to the inputs that affected the output can be retraced. There is little support for actually identifying test cases. Figure 2.13 shows a cause-and-effect graph for the commission problem.



**Figure 2.12 Cause-and-effect graphing operations.**



**Figure 2.13 Cause-and-effect graph for commission problem.**

## 2.20 Guidelines and Observations

As with the other testing techniques, decision table-based testing works well for some applications (such as NextDate) and is not worth the trouble for others (such as the commission problem).

Not surprisingly, the situations in which it works well are those in which a lot of decision making takes place (such as the triangle problem), and those in which important logical relationships exist among input variables (the NextDate function).

1. The decision table technique is indicated for applications characterized by any of the following:
  - a. Prominent if-then-else logic
  - b. Logical relationships among input variables
  - c. Calculations involving subsets of the input variables
  - d. Cause-and-effect relationships between inputs and outputs
  - e. High cyclomatic complexity

2. Decision tables do not scale up very well (a limited entry table with  $n$  conditions has  $2^n$  rules). There are several ways to deal with this—use extended entry decision tables, algebraically simplify tables, “factor” large tables into smaller ones, and look for repeating patterns of condition entries. Try factoring the extended entry table for NextDate (Table 2.27).
3. As with other techniques, iteration helps. The first set of conditions and actions you identify may be unsatisfactory. Use it as a stepping stone and gradually improve on it until you are satisfied with a decision table.

## **Fault-Based Testing**

Fault-based testing uses a fault model directly to hypothesize potential faults in a program under test, as well as to create or evaluate test suites based on its efficacy in detecting those hypothetical faults.

A model of potential program faults is a valuable source of information for evaluating and designing test suites. Some fault knowledge is commonly used in functional and structural testing, for example when identifying singleton and error values for parameter characteristics in category- partition testing or when populating catalogs with erroneous values, but a fault model can also be used more directly. Fault-based testing uses a fault model directly to hypothesize potential faults in a program under test, as well as to create or evaluate test suites based on its efficacy in detecting those hypothetical faults.

## **Overview**

Engineers study failures to understand how to prevent similar failures in the future. For example, failure of the Tacoma Narrows Bridge in 1940 led to new understanding of oscillation in high wind and to the introduction of analyses to predict and prevent such destructive oscillation in subsequent bridge design. The causes of an airline crash are likewise extensively studied, and when traced to a structural failure they frequently result in a directive to apply diagnostic tests to all aircraft considered potentially vulnerable to similar failures. Experience with common software faults sometimes leads to improvements in design methods and programming languages. For example, the main purpose of automatic memory management in Java is not to spare the programmer the trouble of releasing unused memory, but to prevent the programmer from making the kind of memory management errors (dangling pointers, redundant deallocations, and memory leaks) that frequently occur in C and C++ programs. Automatic array bounds checking cannot prevent a programmer from using an index expression outside array bounds, but can make it much less likely that the fault escapes detection in testing, as well as limiting the damage incurred if it does lead to operational failure (eliminating, in particular, the buffer overflow attack as a means of subverting privileged programs). Type checking reliably detects many other faults during program translation.

## Assumptions in Fault-Based Testing

The effectiveness of fault-based testing depends on the quality of the fault model and on some basic assumptions about the relation of the seeded faults to faults that might actually be present. In practice, the seeded faults are small syntactic changes, like replacing one variable reference by another in an expression, or changing a comparison from  $<$  to  $<=$ . We may hypothesize that these are representative of faults actually present in the program.

Put another way, if the program under test has an actual fault, we may hypothesize that it differs from another, corrected program by only a small textual change. If so, then we need merely distinguish the program from all such small variants (by selecting test cases for which either the original or the variant program fails) to ensure detection of all such faults. This is known as the competent programmer hypothesis, an assumption that the program under test is "close to" (in the sense of textual difference) a correct program.

## Fault-Based Testing: Terminology

**Original program** The program unit (e.g., C function or Java class) to be tested.

**Program location** A region in the source code. The precise definition is defined relative to the syntax of a particular programming language. Typical locations are statements, arithmetic and Boolean expressions, and procedure calls.

**Alternate expression** Source code text that can be legally substituted for the text at a program location. A substitution is legal if the resulting program is syntactically correct (i.e., it compiles without errors).

**Alternate program** A program obtained from the original program by substituting an alternate expression for the text at some program location.

**Distinct behavior of an alternate program  $R$  for a test  $t$**  The behavior of an alternate program  $R$  is distinct from the behavior of the original program  $P$  for a test  $t$ , if  $R$  and  $P$  produce a different result for  $t$ , or if the output of  $R$  is not defined for  $t$ .

**Distinguished set of alternate programs for a test suite  $T$**  A set of alternate programs are distinct if each alternate program in the set can be distinguished from the original program by at least one test in  $T$ .

Fault-based testing can guarantee fault detection only if the competent programmer hypothesis and the coupling effect hypothesis hold. But guarantees are more than we expect from other approaches to designing or evaluating test suites, including the structural and functional test adequacy criteria discussed in earlier chapters. Fault-based testing techniques can be useful even if we decline to take the leap of faith required to fully accept their underlying assumptions.

What is essential is to recognize the dependence of these techniques, and any inferences about software quality based on fault-based testing, on the quality of the fault model. This also implies that developing better fault models, based on hard data about real faults rather than guesses, is a good investment of effort.

## Mutation Analysis

Mutation analysis is the most common form of software fault-based testing. A fault model is used to produce hypothetical faulty programs by creating variants of the program under test. Variants are created by "seeding" faults, that is, by making a small change to the program under test following a pattern in the fault model. The patterns for changing program text are called mutation operators, and each variant program is called a mutant.

### Mutation Analysis: Terminology

**Original program under test** The program or procedure (function) to be tested.

**Mutant** A program differs the original program for one syntactic element (e.g., a statement, a condition, a variable, a label).

**Distinguished mutant** A mutant that can be distinguished for the original program by executing at least one test case.

**Equivalent mutant** A mutant that cannot be distinguished from the original program.

**Mutation operator** A rule for producing a mutant program by syntactically modifying the original program. Mutants should be plausible as faulty programs. Mutant programs that are rejected by a compiler, or that fail almost all tests, are not good models of the faults we seek to uncover with systematic testing.

by substituting while for switch in the statement at line 13 would not be valid, since it would result in a compile-time error. We say a mutant is useful if, in addition to being valid, its behavior differs from the behavior of the original program for no more than a small subset of program test cases. A mutant obtained by substituting 0 for 1000 in the statement at line 4 would be valid, but not useful, since the mutant would be distinguished from the program under test by all inputs and thus would not give any useful information on the effectiveness of a test suite. Defining mutation operators that produce valid and useful mutations is a nontrivial task.



```
1
2 /** Convert each line from standard input */
3 void transduce() {
4 #define BUFLen 1000
5 char buf[BUFLen]; /* Accumulate line into this buffer */
6 int pos=0; /* Index for next character in buffer */
7
8 char inChar; /* Next character from input */
9
10 int atCR = 0; /* 0="within line", 1="optional DOS LF" */
11
12 while ((inChar = getchar()) != EOF ) {
13 switch (inChar) {
14 case LF:
15 if (atCR) { /* Optional DOS LF */
16 atCR = 0;
17 } else { /* Encountered CR within line */
18 emit(buf, pos);
19 pos=0;
20 }
21 break;
22 case CR:
23 emit(buf, pos);
24 pos=0;
25 atCR = 1;
26 break;
27 default:
28 if (pos >= BUFLen-2) fail("Buffer overflow");
29 buf[pos++] = inChar;
30 } /* switch */
31 }
32 if (pos > 0) {
33 emit(buf, pos);
34 }
35 }
```

**Figure 2.14: Program transduces converts line endings among Unix, DOS, and Macintosh conventions. The main procedures, which selects the output line end convention and the output procedure emit are not shown.**

Since mutants must be valid, mutation operators are syntactic patterns defined relative to particular programming languages. Figure 2.15 shows some mutation operators for the C language. Constraints are associated with mutation operators to guide selection of test cases likely to distinguish mutants from the original program. For example, the mutation operator svr (scalar variable replacement) can be applied only to variables of compatible type (to be valid), and a test case that distinguishes the mutant from the original program must execute the modified statement in a state in which the original variable and its substitute have different values.

| ID                              | Operator                                 | Description  | Constraint                       |
|---------------------------------|--|--|----------------------------------|
| <i>Operand Modifications</i>    |  |  |                                  |
| crp                             | Constant for constant replacement        | Replace constant C1 with constant C2                               | $C1 \neq C2$                     |
| scr                             | Scalar for constant replacement          | Replace constant C with scalar variable X                          | $C \neq X$                       |
| acr                             | Array for constant replacement           | Replace constant C with array reference A[I]                       | $C \neq A[I]$                    |
| scr                             | Struct for constant replacement          | Replace constant C with struct field S                             | $C \neq S$                       |
| svr                             | Scalar variable replacement              | Replace scalar variable X with a scalar variable Y                 | $X \neq Y$                       |
| cstr                            | Constant for scalar variable replacement | Replace scalar variable X with a constant C                        | $X \neq C$                       |
| asr                             | Array for scalar variable replacement    | Replace scalar variable X with an array reference A[I]             | $X \neq A[I]$                    |
| ssr                             | Struct for scalar replacement            | Replace scalar variable X with struct field S                      | $X \neq S$                       |
| cnr                             | Comparable array replacement             | Replace array reference with a comparable array reference          |                                  |
| sar                             | Struct for array reference replacement   | Replace array reference A[I] with a struct field S                 | $A[I] \neq S$                    |
| <i>Expression Modifications</i> |  |  |                                  |
| abs                             | Absolute value                           | Replace e by abs(e)  | $e < 0$                          |
| aor                             | Arithmetic operator replacement          | Replace arithmetic operator $\psi$ with arithmetic operator $\phi$ | $e_1 \psi e_2 \neq e_1 \phi e_2$ |
| lcr                             | Logical connector replacement            | Replace logical connector $\psi$ with logical connector $\phi$     | $e_1 \psi e_2 \neq e_1 \phi e_2$ |
| <i>Statement Modifications</i>  |  |  |                                  |
| sdl                             | Statement deletion                       | Delete a statement   |                                  |
| sca                             | Switch case replacement                  | Replace the label of one case with another                         |                                  |
| ses                             | End block shift                          | Move ) one statement earlier and later                             |                                  |

**Figure 2.15:** A sample set of mutation operators for the C language, with associated constraints to select test cases that distinguish generated mutants from the original program.

### Fault-Based Adequacy Criteria

Given a program and a test suite  $T$ , mutation analysis consists of the following steps:

**Select mutation operators** If we are interested in specific classes of faults, we may select a set of mutation operators relevant to those faults.

#### Generate mutants

Mutants are generated mechanically by applying mutation operators to the original program.

**Distinguish mutants** Execute the original program and each generated mutant with the test cases in  $T$ . A mutant is *killed* when it can be distinguished from the original program.

A mutant can remain live for two reasons:-

- ♣ The mutant can be distinguished from the original program, but the test suite  $T$  does not contain a test case that distinguishes them (i.e., the test suite is not adequate with respect to the mutant).
- ♣ The mutant cannot be distinguished from the original program by any test case (i.e., the mutant is equivalent to the original program).

$TS = \{1U, 1D, 2U, 2D, 2M, End, Long\}$

| Open table as spreadsheet | Testcase    | Operator  | Line | Original/Mutant                        | 1U | 1D | 2U | 2D | 2M | End | Long | Mixed |
|---------------------------|-------------|-----------|------|--|----|----|----|----|----|-----|------|-------|
|                           | $M_1$       | ror       | 28   | (pos == BUFLen-2)<br>(pos == BUFLen-2) | -  | -  | -  | -  | -  | -   | -    | -     |
|                           | $M_2$       | ror       | 32   | (pos > 0)<br>(pos == 0)                | -  | x  | x  | x  | x  | -   | -    | -     |
|                           | $M_3$       | odi       | 16   | atCR = 0<br>nothing                    | -  | -  | -  | -  | -  | -   | -    | -     |
|                           | $M_4$       | sur       | 16   | atCR = 0<br>pos = 0                    | -  | -  | -  | -  | -  | -   | -    | x     |
| Open table as spreadsheet | Description | Test case |      | Description                            |    |    |    |    |    |     |      |       |

| Open table as spreadsheet | Test case | Description              | Test case | Description                                     |
|---------------------------|-----------|--------------------------|-----------|---|
| 1U                        |           | One line, Unix line-end  | 2M        | Two lines, Mac line-end                         |
| 1D                        |           | One line, DOS line-end   | End       | Last line not terminated with line-end sequence |
| 2U                        |           | Two lines, Unix line-end | Long      | Very long line (greater than buffer length)     |
| 2D                        |           | Two lines, DOS line-end  | Mixed     | Mix of DOS and Unix line ends in the same file  |

**Figure 2.16: Fault-Based Adequacy Criteria**

Given a set of mutants  $SM$  and a test suite  $T$ , the fraction of nonequivalent mutants killed by  $T$  measures the adequacy of  $T$  with respect to  $SM$ . Unfortunately, the problem of identifying equivalent mutants is undecidable in general, and we could err either by claiming that a mutant is equivalent to the program under test when it is not or by counting some equivalent mutants among the remaining live mutants.

The adequacy of the test suite  $TS$  evaluated with respect to the four mutants of Figure 2.16 is 25%. However, we can easily observe that mutant  $M_1$  is equivalent to the original program (i.e., no input would distinguish it).

Conversely, mutants  $M_k$  and  $M_l$  seem to be nonequivalent to the original program: There should be at least one test case that distinguishes each of them from the original program. Thus the adequacy of  $TS$ , measured after eliminating the equivalent mutant  $M_i$ , is 33%.

### **Mutation Analysis vs. Structural Testing**

For typical sets of syntactic mutants, a mutation-adequate test suite will also be adequate with respect to simple structural criteria such as statement or branch coverage. Mutation adequacy can simulate and subsume a structural coverage criterion if the set of mutants can be killed only by satisfying the corresponding test coverage obligations. Statement coverage can be simulated by applying the mutation operator *sdl* (statement deletion) to each statement of a program. To kill a mutant whose only difference from the program under test is the absence of statement  $S$  requires executing the mutant and the program under test with a test case that executes  $S$  in the original program. Thus to kill all mutants generated by applying the operator *sdl* to statements of the program under test, we need a test suite that causes the execution of each statement in the original program. Branch coverage can be simulated by applying the operator *cpr* (constant for predicate replacement) to all predicates of the program under test with constants *True* and *False*. To kill a mutant that differs from the program under test for a predicate  $P$  set to the constant value *False*, we need to execute the mutant and the program under test with a test case that causes the execution of the *True* branch of  $P$ . To kill a mutant that differs from the program under test for a predicate  $P$  set to the constant value *True*, we need to execute the mutant and the program under test with a test case that causes the execution of the *False* branch of  $P$ .

### **Variations on Mutation Analysis**

1. The mutation analysis process described in the preceding sections, which kills mutants based on the outputs produced by execution of test cases, is known as strong mutation.
2. It can generate a number of mutants quadratic in the size of the program. Each mutant must be compiled and executed with each test case until it is killed.
3. The time and space required for compiling all mutants and for executing all test cases for each mutant may be impractical.

## **Estimating Population Sizes**

### **Counting fish**

Lake Winnemunchie is inhabited by two kinds of fish, a native trout and an introduced species of chub. The Fish and Wildlife Service wishes to estimate the populations to evaluate their efforts to eradicate the chub without harming the population of native trout.

The population of chub can be estimated statistically as follows. 1000 chub are netted, their dorsal fins are marked by attaching a tag, then they are released back into the lake. Over the next weeks, fishermen are asked to report the number of tagged and untagged chub caught. If 50 tagged chub and 300 untagged chub are caught, we can calculate and thus there are about 6000 untagged chub remaining in the lake.

$$\frac{1000}{\text{untagged chub population}} = \frac{50}{300}$$

It may be tempting to also ask fishermen to report the number of trout caught and to perform a similar calculation to estimate the ratio between chub and trout. However, this is valid only if trout and chub are equally easy to catch, or if one can adjust the ratio using a known model of trout and chub vulnerability to fishing.

### Counting residual faults

A similar procedure can be used to estimate the number of faults in a program: Seed a given number  $S$  of faults in the program. Test the program with some test suite and count the number of revealed faults. Measure the number of seeded faults detected,  $D_s$ , and also the number of natural faults  $D_N$  detected. Estimate the total number of faults remaining in the program, assuming the test suite is as effective at finding natural faults as it is at finding seeded faults, using the formula

$$\frac{S}{\text{total natural faults}} = \frac{D_s}{D_N}$$

If we estimate the number of faults remaining in a program by determining the proportion of seeded faults detected, we must be wary of the pitfall of estimating trout population by counting chub. The seeded faults are chub, the real faults are trout, and we must either have good reason for believing the seeded faults are no easier to detect than real remaining faults, or else make adequate allowances for uncertainty. The difference is that we cannot avoid the problem by repeating the process with trout - once a fault has been detected, our knowledge of its presence cannot be erased. We depend, therefore, on a very good fault model, so that the chub are as representative as possible of trout. Of course, if we use special bait for chub, or design test cases to detect particular seeded faults, then statistical estimation of the total population of fish or errors cannot be justified.

## Hardware Fault-based Testing

Fault-based testing is widely used for semiconductor and hardware system validation allowances for uncertainty. The difference is that we cannot avoid the problem by repeating the process with trout - once a fault has been detected, our knowledge of its presence cannot be erased. We depend, therefore, on a very good fault model, so that the chub are as representative as possible of trout. Of course, if we use special bait for chub, or design test cases to detect particular seeded faults, then statistical estimation of the total population of fish or errors cannot be justified.

and evaluation both for evaluating the quality of test suites and for evaluating fault tolerance.

Semiconductor testing has conventionally been aimed at detecting random errors in fabrication, rather than design faults. Relatively simple fault models have been developed for testing semiconductor memory devices, the prototypical faults being "stuck-at-0" and "stuck-at-1" (a gate, cell, or pin that produces the same logical value regardless of inputs). A number of more complex fault models have been developed for particular kinds of semiconductor devices (e.g., failures of simultaneous access in dualport memories). A test vector (analogous to a test suite for software) can be judged by the number of hypothetical faults it can detect, as a fraction of all possible faults under the model.

In evaluation of fault tolerance in hardware, the usual approach is to modify the state or behavior rather than the system under test. Due a difference in terminology between hardware and software , the corruption of state or modification of behavior is called a "fault," and artificially introducing it is called "fault injection." Pin-level fault injection consists of forcing a stuck-at-0, a stuck-at-1, or an intermediate voltage level (a level that is neither a logical 0 nor a logical 1) on a pin of a semiconductor device. Heavy ion radiation is also used to inject random faults in a running system. A third approach, growing in importance as hardware complexity increases, uses software to modify the state of a running system or to simulate faults in a running simulation of hardware logic design. Fault seeding can be used statistically in another way: To estimate the number of faults remaining in a program. Usually we know only the number of faults that have been detected, and not the number that remains

















