Chapter 6

Equivalence Class Testing

The use of equivalence classes as the basis for functional testing has two motivations: we would like to have a sense of complete testing, and, at the same time, we would hope to avoid redundancy. Neither of these hopes is realized by boundary value testing—looking at the tables of test cases, it is easy to see massive redundancy, and looking more closely, serious gaps exist. Equivalence class testing echoes the two deciding factors of boundary value testing, robustness and the single/multiple fault assumption. This chapter presents the traditional view of equivalence class testing, followed by a coherent treatment of four distinct forms based on the two assumptions. The single versus multiple fault assumption yields the weak/strong distinction and the focus on invalid data yields a second distinction: normal versus robust. Taken together, these two assumptions result in Weak Normal, Strong Normal, Weak Robust, and Strong Robust Equivalence Class testing.

Two problems occur with robust forms. The first is that, very often, the specification does not define what the expected output for an invalid input should be. (We could argue that this is a deficiency of the specification, but that does not get us anywhere.) Thus, testers spend a lot of time defining expected outputs for these cases. The second problem is that strongly typed languages eliminate the need for the consideration of invalid inputs. Traditional equivalence testing is a product of the time when languages such as FORTRAN and COBOL were dominant; thus, this type of error was common. In fact, it was the high incidence of such errors that led to the implementation of strongly typed languages.

6.1 Equivalence Classes

In Chapter 3, 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, the union of which 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 disjointedness ensures a form of nonredundancy. 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 code-based testing in Chapters 8 and 9, we shall see that "treated the same" maps onto "traversing the same execution path." The four forms of equivalence class testing all address the problems of gaps and redundancies that are common to the four forms of boundary value testing. Since the assumptions align, the four forms of boundary value testing also align with the four forms of equivalence class testing. There will be one point of overlap—this occurs when equivalence classes are defined by bounded variables. In such cases, a hybrid of boundary value and equivalence class testing is appropriate. The International Software Testing Qualifications Board (ISTQB) syllabi refer to this as "edge testing." We will see this in the discussion in Section 6.3.

6.2 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 6.1 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 6.1 shows the five test cases for this process for our continuing function F of two variables.

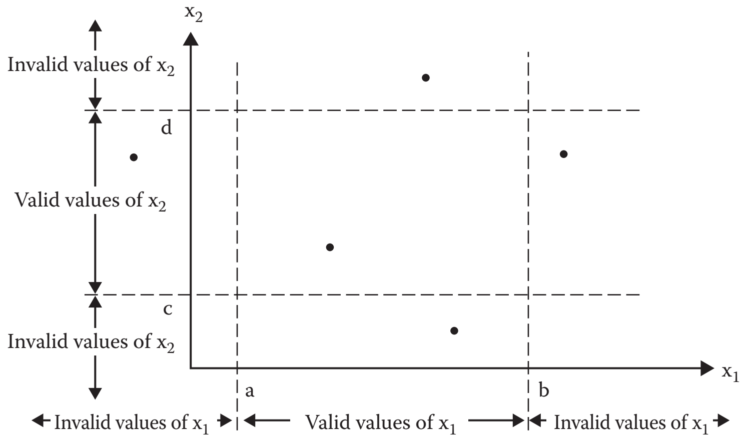

Figure 6.1 Traditional equivalence class test cases.

6.3 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:

$$a \le x_1 \le d, \text{ with intervals } [a, b), [b, c), [c, d]$$
$$e \le x_2 \le g, \text{ with intervals } [e, f), [f, g]$$

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

$V1 = \{x_1: a \le x_1 < b\}$, $V2 = \{x_1: b \le x_1 < c\}$, $V3 = \{x_1: c \le x_1 \le d\}$, $V4 = \{x_2: e \le x_2 < f\}$, $V5 = \{x_2: f \le x_2 \le g\}$

The equivalence classes of invalid values are

$$NV1 = \{x_1: x_1 < a\}, NV2 = \{x_1: d < x_1\}, NV3 = \{x_2: x_2 < e\}, NV4 = \{x_2: g < x_2\}$$

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.

6.3.1 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 6.2. 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 x_1 in the class [a, b), and to a value of x_2 in the class [e, f). The test case in the upper center rectangle corresponds to a value of x_1 in the class [b, c) and to a value of x_2 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 x_1, or a problem with x_2, 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.

6.3.2 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 6.3. 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.

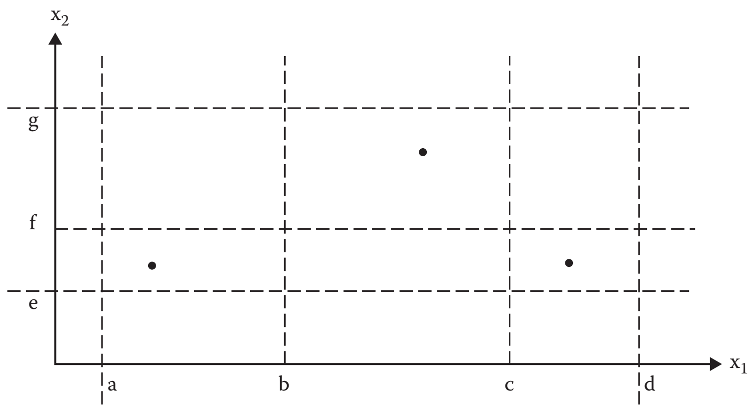

Figure 6.2 Weak normal equivalence class test cases.

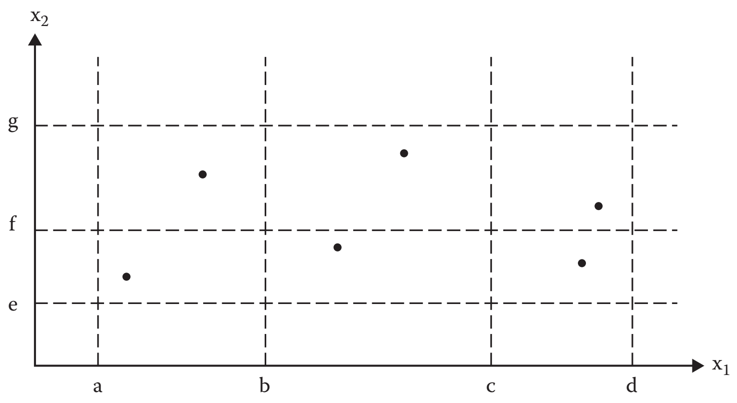

Figure 6.3 Strong normal equivalence class test cases.

6.3.3 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 6.4, the test cases for valid classes are as those in Figure 6.2. 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.)

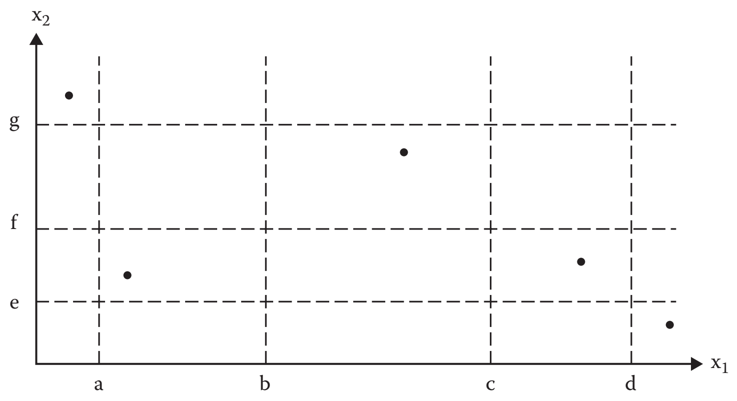

Figure 6.4 Weak robust equivalence class test cases.

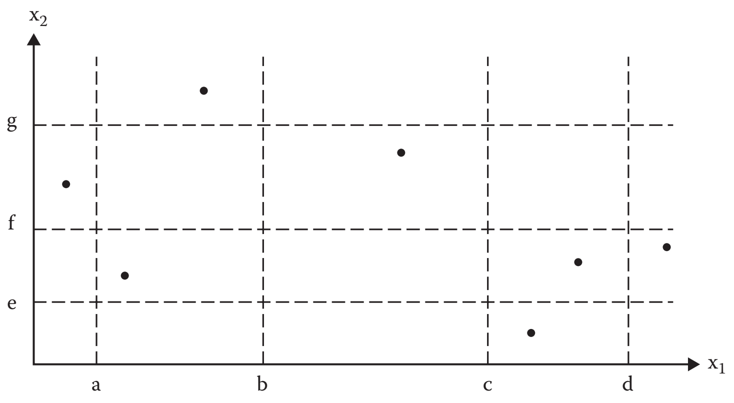

Figure 6.5 Revised weak robust equivalence class test cases.

The test cases resulting from this strategy are shown in Figure 6.4. 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 6.5 presents a compromise between "pure" weak normal equivalence class testing and its robust extension.

6.3.4 Strong Robust Equivalence Class Testing

At least the name for this form is neither counterintuitive nor oxymoronic, just redundant. As before, the robust part comes from consideration of invalid values, and the strong part refers to the multiple fault assumption. We obtain test cases from each element of the Cartesian product of all the equivalence classes, both valid and invalid, as shown in Figure 6.6.

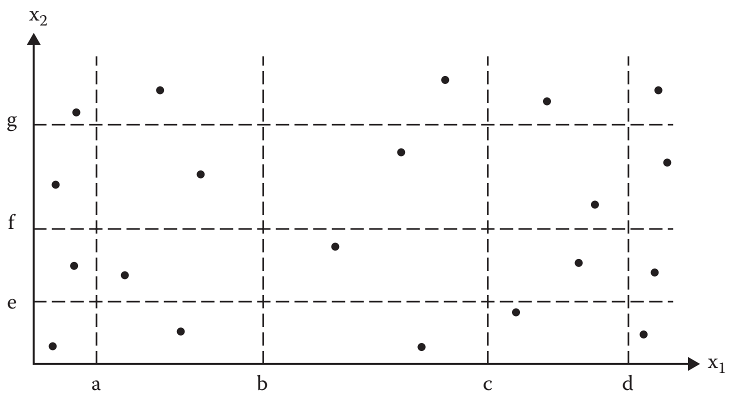

Figure 6.6 Strong robust equivalence class test cases.

6.4 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:

Test Case	a	b	c	*Expected Output*
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′ = {<a, b, c>: a = b + c}
D6″ = {<a, b, c>: a > b + c}

and similarly for D7 and D8.

6.5 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 = {month: $1 \le \text{month} \le 12$}
D1 = {day: $1 \le \text{day} \le 31$}
Y1 = {year: $1812 \le \text{year} \le 2012$}

The invalid equivalence classes are

M2 = {month: month < 1}
M3 = {month: month > 12}
D2 = {day: day < 1}
D3 = {day: day > 31}
Y2 = {year: year < 1812}
Y3 = {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

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

Case ID	*Month*	*Day*	*Year*	*Expected Output*
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 Value of day not in the range 1 ... 31
SR5	6	–1	1811	Value of day not in the range 1 ... 31 Value of year not in the range 1812 ... 2012
SR6	–1	15	1811	Value of month not in the range 1 ... 12 Value of year not in the range 1812 ... 2012
SR7	–1	–1	1811	Value of month not in the range 1 ... 12 Value of day not in the range 1 ... 31 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 ≤ day ≤ 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.

6.5.1 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:

Case ID	*Month*	*Day*	*Year*	*Expected Output*
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:

Case ID	*Month*	*Day*	*Year*	*Expected Output*
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

Case ID	*Month*	*Day*	*Year*	*Expected Output*
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

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.

6.6 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 ≤ locks ≤ 70}
L2 = {locks = –1} (occurs if locks = –1 is used to control input iteration)
S1 = {stocks: 1 ≤ stocks ≤ 80}
B1 = {barrels: 1 ≤ barrels ≤ 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.

Case ID	*Locks*	*Stocks*	*Barrels*	*Expected Output*
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:

Case ID	*Locks*	*Stocks*	*Barrels*	*Expected Output*
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 Value of stocks not in the range 1 ... 80
SR5	–2	40	–1	Value of locks not in the range 1 ... 70 Value of barrels not in the range 1 ... 90
SR6	35	–1	–1	Value of stocks not in the range 1 ... 80 Value of barrels not in the range 1 ... 90
SR7	–2	–1	–1	Value of locks not in the range 1 ... 70 Value of stocks not in the range 1 ... 80 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:

Sales = 45 <mathMultiply> locks + 30 <mathMultiply> stocks + 25 <mathMultiply> barrels

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

S1 = {<locks, stocks, barrels>: sales ≤ 1000}
S2 = {<locks, stocks, barrels>: 1000 < sales ≤ 1800}
S3 = {<locks, stocks, barrels>: sales > 1800}

Figure 5.6 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 5.6.

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.

Test Case	*Locks*	*Stocks*	*Barrels*	*Sales*	*Commission*
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.

6.7 Edge Testing

The *ISTQB Advanced Level Syllabus* (ISTQB, 2012) describes a hybrid of boundary value analysis and equivalence class testing and gives it the name "edge testing." The need for this occurs when contiguous ranges of a particular variable constitute equivalence classes. Figure 6.2 shows three equivalence classes of valid values for x_1 and two classes for x_2. Presumably, these classes refer to variables that are "treated the same" in some application. This suggests that there may be faults near the boundaries of the classes, and edge testing will exercise these potential faults. For the example in Figure 6.2, a full set of edge testing test values are as follows:

Normal test values for x_1: {a, a+, b–, b, b+, c–, c, c+, d–, d}
Robust test values for x_1: {a–, a, a+, b–, b, b+, c–, c, c+, d–, d, d+}
Normal test values for x_2: {e, e+, f–, f, f+, g–, g}
Robust test values for x_2: {e–, e, e+, f–, f, f+, g–, g, g+}

One subtle difference is that edge test values do not include the nominal values that we had with boundary value testing. Once the sets of edge values are determined, edge testing can follow any of the four forms of equivalence class testing. The numbers of test cases obviously increase as with the variations of boundary value and equivalence class testing.

6.8 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. (The decision table technique in Chapter 7 resolves this problem.)
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.

EXERCISES

1. Starting with the 36 strong normal equivalence class test cases for the NextDate function, revise the day classes as discussed, and then find the other nine test cases.
2. If you use a compiler for a strongly typed language, discuss how it would react to robust equivalence class test cases.
3. Revise the set of weak normal equivalence classes for the extended triangle problem that considers right triangles.
4. Compare and contrast the single/multiple fault assumption with boundary value and equivalence class testing.
5. The spring and fall changes between standard and daylight savings time create an interesting problem for telephone bills. In the spring, this switch occurs at 2:00 a.m. on a Sunday morning (late March, early April) when clocks are reset to 3:00 a.m. The symmetric change takes place usually on the last Sunday in October, when the clock changes from 2:59:59 back to 2:00:00.

   Develop equivalence classes for a long-distance telephone service function that bills calls using the following rate structure:

   Call duration ≤20 minutes charged at \$0.05 per minute or fraction of a minute
   Call duration >20 minutes charged at \$1.00 plus \$0.10 per minute or fraction of a minute in excess of 20 minutes.

   Make these assumptions:
   - Chargeable time of a call begins when the called party answers, and ends when the calling party disconnects.
   - Call durations of seconds are rounded up to the next larger minute.
   - No call lasts more than 30 hours.
6. If you did exercise 8 in Chapter 2, and exercise 5 in Chapter 5, you are already familiar with the CRC Press website for downloads (http://www.crcpress.com/product/isbn/97818466560680). There you will find an Excel spreadsheet named specBasedTesting.xls.

(It is an extended version of Naive.xls, and it contains the same inserted faults.) Different sheets contain strong, normal equivalence class test cases for the triangle, NextDate, and commission problems, respectively. Run these sets of test cases and compare the results with your naive testing from Chapter 2 and your boundary value testing from Chapter 5.

References

ISTQB Advanced Level Working Party, *ISTQB Advanced Level Syllabus*, 2012.
Mosley, D.J., *The Handbook of MIS Application Software Testing,* Yourdon Press, Prentice Hall, Englewood Cliffs, NJ, 1993.
Myers, G.J., *The Art of Software Testing,* Wiley Interscience, New York, 1979.