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. Three forms of equivalence class testing were identified in the first edition of this book; here, we identify four. The single versus multiple fault assumption yields the weak/strong distinction made in the first edition. The focus on invalid data yields a new distinction: robust versus normal.

Most of the standard testing texts (Myers, 1979; Mosley, 1993) discuss what we will call weak robust equivalence class testing. 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. The usual response to this problem 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. 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.

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 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.

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. Invalid values of x_1 and x_2 are $x_1 < a$, $x_1 > d$ and $x_2 < e$, $x_2 > g$.

6.1.1 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 previous example, we would end up with the weak equivalence class test cases shown in Figure 6.1.

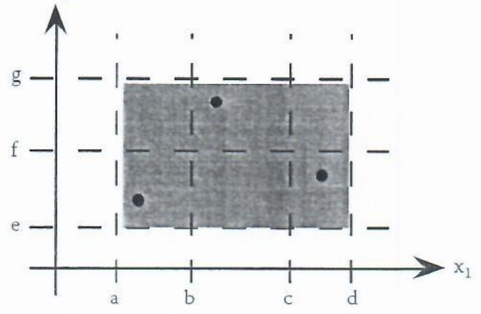

Figure 6.1 Weak normal equivalence class test cases.

These three test cases use one value from each equivalence class. We identify 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.

6.1.2 *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 6.2.

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.

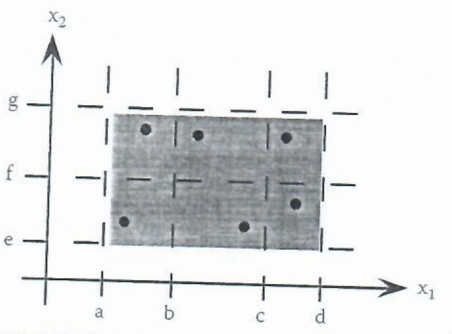

Figure 6.2 Strong normal equivalence class test cases.

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.

6.1.3 *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. (This form was referred to as "traditional equivalence class testing" in the first edition of this book.)

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.)

The test cases resulting from this strategy are shown in Figure 6.3.

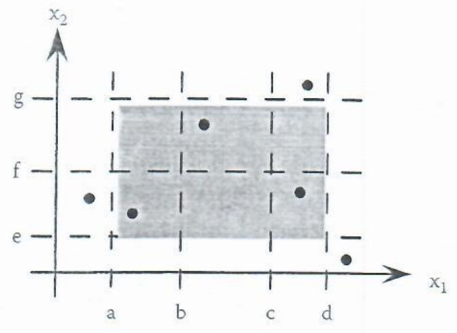

Figure 6.3 Weak robust equivalence class test cases.

Two problems occur with robust equivalence testing. 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.

6.1.4 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. (This form was omitted in the first edition of this book.)

We obtain test cases from each element of the Cartesian product of all the equivalence classes, as shown in Figure 6.4.

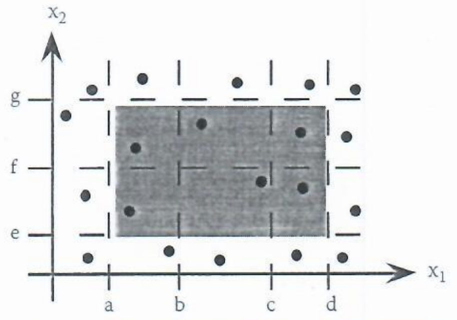

Figure 6.4 Strong robust equivalence class test cases.

6.2 Equivalence Class Test Cases for the Triangle Problem

In the problem statement, we note that four possible outputs can occur: Not a Triangle, 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:

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. SN

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 3-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 input 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 even more thorough, we could separate the "greater than or equal to" into 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.

6.3 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 ≤ month ≤ 12}
D1 = {day : 1 ≤ day ≤ 31}
Y1 = {year : 1812 ≤ year ≤ 2012}

one class for each variable
SN, $1 * 1 * 1 = 1$ = WN

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 3-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:

WN = 4 = biggest set
SW = 3 × 4 × 3 = 36

M1 = {month : month has 30 days}
M2 = {month : month has 31 days}
M3 = {month : month is February}
D1 = {day : $1 \leq$ 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.

will be
ents in
is that
ı more

e func-
month
year is
ıterest-

he last
ıp year
ıented,
ırs: the
ralence

6.3.1 Equivalence Class Test Cases

These classes yield the following weak 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:

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
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

7/1

(continued from previous page)

Case ID	Month	Day	Year	Expected Output
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	Invalid Input Date
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.

6.4 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	$= \{\text{locks} : 1 \le \text{locks} \le 70\}$
L2	$= \{\text{locks} = -1\}$ (occurs if locks = –1 is used to control input iteration)
S1	$= \{\text{stocks} : 1 \le \text{stocks} \le 80\}$
B1	$= \{\text{barrels} : 1 \le \text{barrels} \le 90\}$

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 are 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.

(WN & SW

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

Finally, a corner of the cube will be 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

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

6.4.1 Output Range Equivalence Class Test Cases

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:

$S1 = \{<\text{locks, stocks, barrels}> : \text{sales} \leq 1000\}$
$S2 = \{<\text{locks, stocks, barrels}> : 1000 < \text{sales} \leq 1800\}$
$S3 = \{<\text{locks, stocks, barrels}> : \text{sales} > 1800\}$

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 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