Chapter 8

Retrospective on Functional Testing

In the preceding three chapters, we studied as many types of functional testing. The common thread among these is that all view a program as a mathematical function that maps its inputs onto its outputs. With the boundary-based approaches, test cases are identified in terms of the boundaries of the ranges of the input variables, and variations give us four techniques—boundary value analysis, robustness testing, worst-case testing, and robust worst-case testing. We then took a closer look at the input variables, defining equivalence classes in terms of values that should receive "similar treatment" from the program tested. Four forms of equivalence class testing are used—weak normal, strong normal, weak robust, and strong robust. The goal of examining similar treatment is to reduce the sheer number of test cases generated by the boundary-based techniques. We pushed this a step further when we used decision tables to analyze the logical dependencies imposed by the function of the program. Whenever we have a choice among alternatives, we naturally want to know which is preferred—or at least how to make an informed choice. In this chapter, we look at questions about testing effort and efficiency, and then try to get a handle on test effectiveness.

8.1 Testing Effort

Let us return to our craftsperson metaphor for a minute. We usually think of such people as knowing their crafts so well that their time is spent very effectively. Even if it takes a little longer, we like to think that the time is well spent. We are finally in a position to see a hint of this as far as testing techniques are concerned. The functional methods we have studied vary in terms of both the number of test cases generated and the effort to develop these test cases. Figure 8.1 and Figure 8.2 show the general trends, but the sophistication axis needs some explanation.

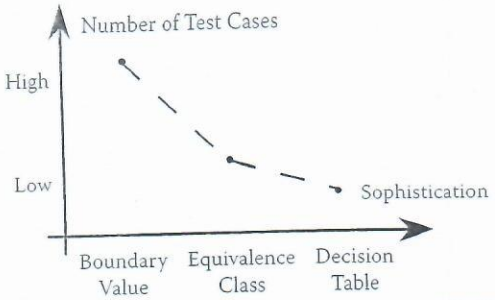

Figure 8.1 Trendline of test cases per testing method.

The boundary-based techniques have no recognition of data or logical dependencies; they are very mechanical in the way they generate test cases. Because of this, they are also easy to automate. The equivalence class techniques pay attention to data dependencies and to the function itself. More thought is required to use these techniques—also more judgment, or craft. The thinking goes into the identification of the equivalence classes; after that, the process is also mechanical. The decision table technique is the most sophisticated, because it requires the tester to consider both data and logical dependencies. As we saw in our examples, you might not get the conditions of a decision table right on the first try, but once you have a good set of conditions, the resulting test cases are both complete and, in some sense, minimal.

The end result is a satisfying trade-off between test identification effort and test execution effort: methods that are easy to use generate numerous test cases, which in turn are more time-consuming to execute. If we shift our effort toward more sophisticated testing methods, we are repaid with less test execution time. This is particularly important because tests are typically executed several times. We might also note that judging testing quality in terms of the sheer number of test cases has drawbacks similar to those of judging programming productivity in terms of lines of code.

Our examples bear out the trends of Figure 8.1 and Figure 8.2. The following three graphs (Figure 8.2 to Figure 8.5) are taken from a spreadsheet that summarized the number of test cases for each of our examples in terms of the various methods. The boundary-based numbers are identical,

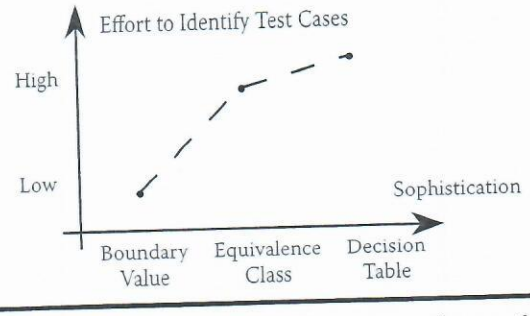

Figure 8.2 Trendline of test case identification effort per testing method.

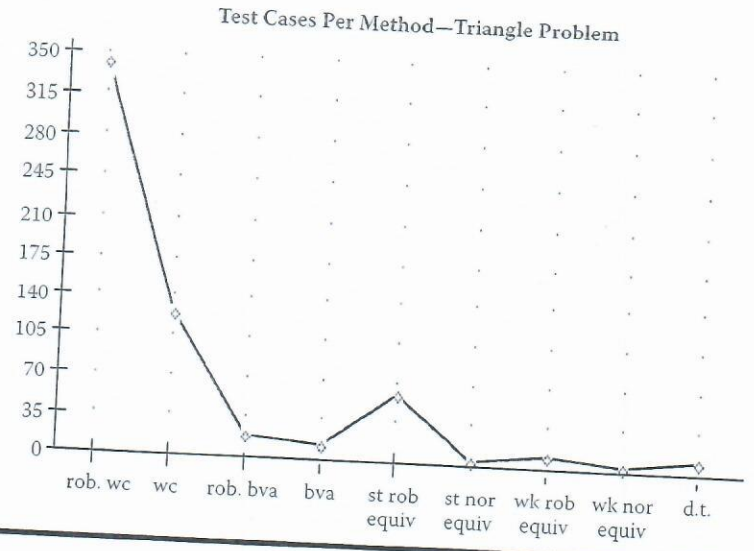

Figure 8.3 Test case trendline for the triangle problem.

reflecting both the mechanical nature of the techniques and the formulas that describe the number of test cases generated by each method. The main differences are seen in strong equivalence class testing and decision table testing. Both of these reflect the logical complexity of the problems, so we would expect to see differences here. When we study structural testing (Chapters 9 and 10), we will see that these distinctions have an important implication for testing. The three graphs are superimposed on each other in Figure 8.6.

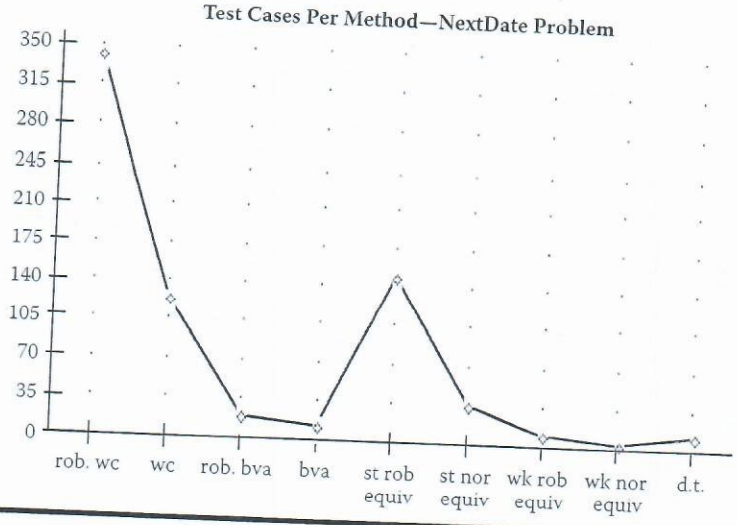

Figure 8.4 Test case trendline for the NextDate problem.

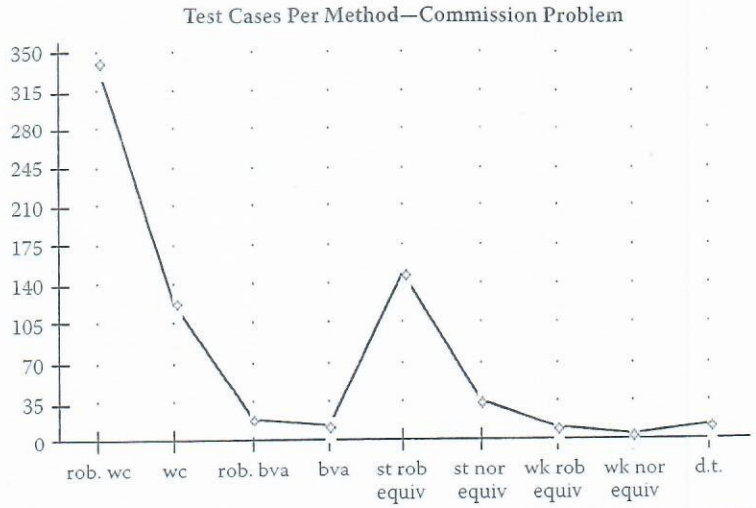

Figure 8.5 Test case trendline for the commission problem.

8.2 Testing Efficiency — Minimum/No Gap/redundancy

If you look closely at these sets of test cases, you can get a feel for the fundamental limitation of functional testing: the twin possibilities of gaps of untested functionality and redundant tests. Consider the NextDate problem, for example. The decision table (which took three tries to get it right) yields 13 test cases. We have confidence that these test cases are complete (in some sense) because the decision table is complete. On the other hand, worst-case boundary value analysis yielded 125 test cases. When we look closely at these, they are not very satisfactory: What do we expect to learn from cases 1 to 5? These test cases check the NextDate function for January 1 in

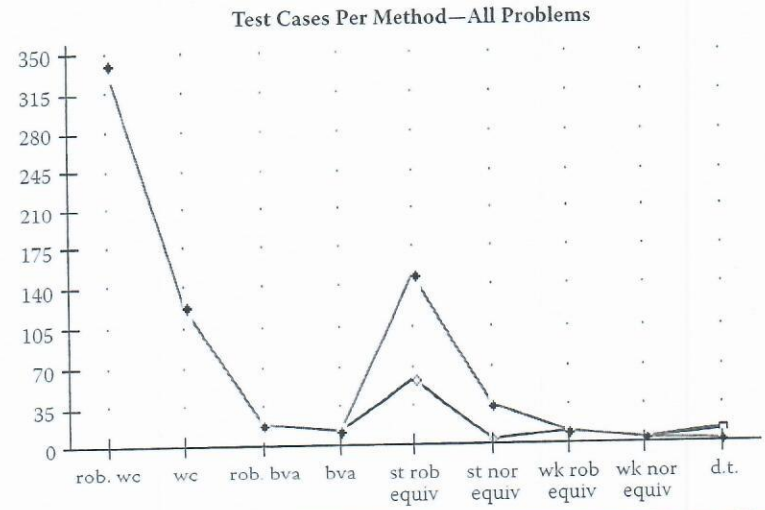

Figure 8.6 Test case trendline for all three problems.

five different years. The year has nothing to do with this part of the calendar, so we would expect that one of these would suffice. If we roughly estimate a "times 10" redundancy, we might expect a reduction to 25 test cases, quite compatible with the 22 from the decision table approach. Looking closer, we find a few cases for February, but none happen to hit February 28 or 29, and no interesting connection is made with leap years. Not only do we have a times 10 redundancy, but we also have some severe gaps around the end of February and leap years.

The strong normal equivalence class test cases move in the right direction: 36 test cases, of which 11 are impossible. Once again, the impossible test cases result from the presumed independence among the equivalence classes. All but six of the decision table test cases (cases 2, 7, 12, 15, 17, and 18 of Table 7.16) map to corresponding strong equivalence class test cases (see Section 6.3.1). Half of these deal with the 28th day of non-February months, so they are not very interesting. The remaining three are useful test cases, especially because they test possibilities that are missed by the strong equivalence class test cases. All this supports two conclusions: gaps occur in the functional test cases, and these gaps are reduced by using more sophisticated techniques.

Can we push this a little further by trying to quantify what we mean by testing efficiency? The intuitive notion is that a set of test cases is just right—that is, no gaps and no redundancy. We can develop various ratios of total number of test cases generated by method A to those generated by method B, or even ratios on a test case basis. This is usually more trouble than it is worth, but sometimes management demands numbers even when they have little real meaning. We will revisit this in Chapter 11, after we complete our study of structural testing. The structural approaches support interesting (and useful) metrics, and these will provide a much better quantification of testing efficiency. Meanwhile, we can help recognize redundancy by annotating test cases with a brief purpose comment. When we see several test cases with the same purpose, we (correctly) sense redundancy. Detecting gaps is harder: If we can only use functional testing, then the best we can do is compare the test cases that result from two methods. In general, the more sophisticated methods will help us recognize gaps with respect to the specification, but nothing is guaranteed. We could develop excellent strong equivalence classes for a program, and then produce a klutzy decision table.

8.3 Testing Effectiveness

What we would really like to know about a set of test cases is how effective they are, but we need to clarify what "effective" means. The easy choice is to be dogmatic: mandate a method, use it to generate test cases, and then run the test cases. This is absolute, and conformity is measurable, so it can be used as a basis for contractual compliance. We can improve on this by relaxing a dogmatic mandate and requiring that testers choose appropriate methods, using the guidelines given at the ends of various chapters here. We can gain another incremental improvement by devising appropriate hybrid methods, as we did with the commission problem in Chapter 5.

Structured testing techniques yield a second choice for test effectiveness. In Chapter 9, we will discuss the notion of program execution paths, which provide a good formulation of test effectiveness. We will be able to examine a set of test cases in terms of the execution paths traversed. When a particular path is traversed more than once, we might question the redundancy. Sometimes such redundancy can have a purpose, as we shall see in Chapter 10.

The best interpretation for testing effectiveness is (no great surprise) the most difficult. We would really like to know how effective a set of test cases is for finding faults present in a program. This is problematic for two reasons: First, it presumes we know all the faults in a program. Quite a

circularity—if we did, we would take care of them. Because we do not know all the faults in a program, we could never know if the test cases from a given method revealed them. The second reason is more theoretical: proving that a program is fault-free is equivalent to the famous halting problem of computer science, which is known to be impossible. The best we can do is to work backward from fault types. Given a particular kind of fault, we can choose testing methods (functional and structural) that are likely to reveal faults of that type. If we couple this with knowledge of the most likely kinds of faults, we end up with a pragmatic approach to testing effectiveness. This is improved if we track the kinds (and frequencies) of faults in the software we develop.

8.4 Guidelines

Here is one of my favorite testing stories. An inebriated man was crawling around on the sidewalk beneath a streetlight. When a policeman asked him what he was doing, he replied that he was looking for his car keys. "Did you lose them here?" the policeman asked. "No, I lost them in the parking lot, but the light is better here."

This little story contains an important message for testers: testing for faults that are not likely to be present is pointless. It is far more effective to have a good idea of the kinds of faults that are most likely (or most damaging) and then to select testing methods that are likely to reveal these faults.

Many times, we do not even have a feeling for the kinds of faults that may be prevalent. What then? The best we can do is use known attributes of the program to select methods that deal with the attributes—sort of a "punishment fits the crime" view. The attributes that are most helpful in choosing functional testing methods are:

Whether the variables represent physical or logical quantities
Whether dependencies exist among the variables
Whether single or multiple faults are assumed
Whether exception handling is prominent

Here is the beginning of an "expert system" on functional testing technique selection:

1. If the variables refer to physical quantities, domain testing and equivalence class testing are indicated.
2. If the variables are independent, domain testing and equivalence class testing are indicated.
3. If the variables are dependent, decision table testing is indicated.
4. If the single fault assumption is warranted, boundary value analysis and robustness testing are indicated.
5. If the multiple fault assumption is warranted, worst-case testing, robust worst-case testing, and decision table testing are indicated.
6. If the program contains significant exception handling, robustness testing and decision table testing are indicated.
7. If the variables refer to logical quantities, equivalence class testing and decision table testing are indicated.

Combinations of these may occur; therefore, the guidelines are summarized as a decision table in Table 8.1.