TESTING AND DEBUGGING

MAXIM - TESTING CANNOT PROVE THERE ARE NO BUGS, IT CAN ONLY HELP FIND SOME OF THEM.

There are two basic ways to try to ensure that the program one writes is correct, that is, that the program performs the way it was intended to perform. The first is to test the program on sample data. In view of the above maxim, one could ask whether this was indeed a fruitful activity. The answer is definitely yes, but adequate testing requires an understanding of the nature of errors and a scientific approach to testing and the development of test data. We will introduce some of the concepts involved in testing and test design. As with general software engineering, whole books have been devoted to just this subarea of software, so we will only be able to give a brief introduction.

1. Errors

It may seem obvious that certain kinds of errors are quite common. Yet, the fact that they are common is an indication that they need to be constantly thought of in writing software. Some categories that cover a large percentage of errors are:

initialization
boundary problems
failure to check for invalid input
off by one
unexpected cases

2. Testing

The purpose of testing is to execute the program in order to find and fix as many errors as possible. In order to do this, it is necessary to design sets of test data that have a high probability of showing up the mistakes in a program. There are two basic approaches to the derivation of test data - function and structural. Functional testing takes <u>no</u> account of the actual code itself, but rather derives test cases from an understanding of the purpose of the software. If a program is supposed to alphabetize lists of names, then various test cases should include lists that have duplicates, lists that include names which are initial substrings of other names (e.g., Smith and Smithe), the empty list of names, etc. None of these has anything to do with the particular sorting algorithm implemented in the program; they just have to do with the purpose of the program. The opposite approach is the structural approach, that is, developing test data based on the actual statements in the program. Thus, if a program contained nested conditionals, test data that forced the execution down all the possible cases and sub-cases should be developed. In general, it is a good idea to do some of each kind of testing.

It is interesting to note that experience suggests it is better for someone who did NOT write the program to do the testing. There are a variety of reasons for this. For one thing, the tester has no preconceived ideas about the code and will therefore be less likely to "overlook" something that is thought to be "obviously ok". For another, the tester is more likely to be impartial and "tough" in testing the program because the tester has no ego involvement with it. Of course, it is not always possible to have a separate person do the testing. However, most software is developed in teams, so it would be quite possible to arrange for a function to be tested at least by someone who didn't write that particular code if not by a separate team of testers altogether.

2.1 Functional Testing

Functional testing is driven by the problem specification and design. It requires an analysis of the nature of the data of the application and does not utilize properties of the actual code. The main aspects of the analysis are to find natural boundaries in the data and to test the program on samples that lie well within the boundaries as well as samples that lie near or on those boundaries. In addition, there should be test cases to verify that the program responds appropriately to invalid inputs when the program is required to do so. Let's consider some examples to see how the boundaries can be determined and why they are important.

Example. A program is to compute averages of student test scores. It should handle up to 200 test scores. Scores are to be in the range 0 to 100 inclusive. For this problem there are two aspects of the underlying data - how many test scores are there and the range of values for test scores. For the first one, the boundaries are 0 (i.e., no test scores at all) and 200 (the maximum allowed number of test scores). Note that the one limit is a natural one not really dependant on the problem specification; it is not possible to have -5 scores in a list. The second one comes directly from the statement of the problem. Based on these remarks, test cases should include an empty list of scores, a list with only 1 score, a list of 199 scores, a list of 200 scores, a list with 201 scores, probably several lists of various lengths between 2 and 198, and perhaps one or two lists with 250 scores or 300 scores. We have suggested test cases for several values at or near the boundaries (e.g., lists of length 0 and of length 1 as well as lists of length 199, 200 and 201) so as to test carefully for "off-by-one" type errors. If the problem says handle up to 200 scores, the program should handle both 199 scores and 200 scores and should behave differently when there are 201 scores. Note that in terms of checking whether the program performs correctly for this problem, averaging a list of 55 scores is probably as good a test as averaging a list of 60 scores. That is, for test cases away from the boundaries, any test case is as good as any other. Now consider the second aspect of the data in this problem, the scores themselves. Here the boundaries are 0 and 100. Moreover, the program should be able to recognize negative inputs as well as scores in the legal range and inputs over 100 and correctly respond to the inputs that are too small or too large. Thus, there should be test cases with negative scores, scores of -1, scores of 0, scores of 1, scores of 99, scores of 100, scores of 101 and scores larger than 101. There can also be interaction between the two aspects, namely the existence of illegal scores can cause the test case to be near a border in terms of number of scores. So we would add cases such as a list of 5 scores that are all illegal (no scores to average) or all but one are illegal (near the boundary) etc. To summarize, then, we have

number of scores:	0, 1 to 200, greater than 200
score values:	negative, 0 to 100, greater than 100

and the test data should include all combinations of situations for these two features.

Reviewing the above example in terms of the general remarks at the beginning of this section, we have analyzed the data of the problem, found certain boundaries in the data and thereby identified boundary areas as well as classes of equivalent test cases that need to be given to the program. We have also identified data validity checks (in this case as a natural consequence of the boundaries for test scores themselves). We can now think of other examples of such boundary and validity analysis:

a. manipulating binary trees - some natural boundaries are
  1. the tree is empty
  2. the tree has just one node
  3. a branch has only left children
  4. a branch has only right children

b. reading lists of alphanumeric tokens of up to 8 characters -
  1. first character is not alphanumeric (validity check)
  2. the next token has only 1 character
  3. the next token has 8 characters

For the token example, one would include test cases where there were 1-character tokens, 7-character tokens, 8-character tokens, strings of 9 characters, strings of, say, 15 characters as well as strings of 2-6 characters (the normally expected situation).

Notice that in all these examples it was not necessary to know any details of the actual code. Of course, we could predict that the program to average up to 200 scores has a conditional statement or a loop that checks for greater than 200, but we really don't make use of that in analyzing the data. The boundaries come directly from the problem statement itself.

Of course, the test cases should be independently verifiable; that is, there should be some means other than the program itself to determine what the output or behavior of the program should be. Often this means working the test cases by hand. In such cases the test data must be relatively simple, but should be as exhaustive as possible or practicable. Also, for a large problem with several different aspects of the data there could be a large number of combinations of test cases from the various classes. Often, several aspects from different classes can be tested in a single run. For example one single test case could be used to check the basic averaging computation for the test-scores example and check score values at all the boarders as well. Also, the use of modularization helps partition the testing into smaller bits - each module can be tested separately, and then data for testing the combination of modules need not be so complicated. Still for a complicated problem there will simply be a lot of test cases to be tried.

2.2 Structural Testing

Structural testing is just the opposite of functional testing; structural testing derives its test cases by direct analysis of the code - where are the control points, what are the actual conditional expressions governing the control points, etc. Full structural testing would guarantee that every statement in the code is "exercised", that every possible outcome of conditional expressions is tried, and eventually that every combination of paths through the program is tried. Of course, the last of these may not be feasible in a larger program; however, the use of structured programming cuts down on the number of path combinations, so it may still be feasible in a module that is not too large. Moreover, as we will see shortly, there are 4 levels of structural testing, and the testing can proceed from the simplest type as far toward the full testing as time and resources allow or the need for correctness demands.

The notion of boundaries again plays a role, except that for structural testing the boundaries are extracted from the code. For example if a program contains a conditional statement with condition j<=200, there should be test cases where j is 199, j is 200 and j is 201 as well as cases where j is far away from 200.

In order to analyze the structure of a program, a device called a program flow diagram is often used. A program flow diagram is a graphical abstraction of the program. It shows the possible paths and branch points in the control scheme of a program without showing the details of the individual statements. More specifically, the diagram contains one vertex or node for each statement (including sub-statements of a block). Then there is an arrow from node j to node k if and only if there are some conditions under which the execution of the program can move to statement k immediately after statement j. It is then common to reduce sequences of edges in the diagram in which there are no branch points or junction points, that is, sequences where there is only one starting point and one ending point and one path from the start to the end. We illustrate with some examples.

EXAMPLE 1.

```
a = a * b + 1;
b = b * b - 4;
if (a > b) cout << a;
if (x > 0) x = x-1;
cin >> y;
```

This segment of code has 5 main statements (two assignments, two if-statements and a cin statement). Each if-statement has a sub-statement. Let's use numbers to represent the nodes corresponding to the various statements:

1 and 2 for the two assignment statements
3 for the first if-statement
4 for the cout statement
5 for the second if-statement

6 for the sub part of the last if-statement
7 for the cin statement.

Statement 1 always flows into statement 2, which in turn always flows into statement 3. At statement 3, there are two possibilities. One is that the if-statement will flow into the cout statement; the other is that it will flow immediately to the next if statement. There are similarly two possibilities for the second if statement. The flow diagram for the above code segment is

$$\begin{tikzcd}
1 \arrow[r] & 2 \arrow[r] & 3 \arrow[r] \arrow[rr, bend left] & 4 \arrow[r] \arrow[rr, bend right] & 5 \arrow[r] & 6 \arrow[r] & 7
\end{tikzcd}$$

The nodes 1 and 2 can be compressed because there are no branch or junction points. The resulting compressed graph is

$$\begin{tikzcd}
(1,2) \arrow[r] & 3 \arrow[r] \arrow[rr, bend left] & 4 \arrow[r] \arrow[rr, bend right] & 5 \arrow[r] & 6 \arrow[r] & 7
\end{tikzcd}$$

EXAMPLE 2.
Consider the following section of code:

```
for (i = 0, i<100, i++)
        { x += y;
          y += i;
        }
```

Here there is one main statement, the for-statement, and two sub-statements. Using 1 for the for statement, 2 for the first assignment statement, 3 for the second assignment statement, and 4 for the statement after the for loop, and noting that 3 always follows from 2, we can form the following flow diagram:

$$\begin{tikzcd}
1 \arrow[r] \arrow[rr, bend right] & (2,3) \arrow[l, bend right] & 4
\end{tikzcd}$$

There are four levels of structural test cases, each testing the code more completely than the previous:

I. Statement coverage - every statement gets executed at least once.

II. Edge coverage/Decision coverage -- every branch is traced, every decision is made in all possible ways.

III. Condition coverage -- all subparts of a condition are tested.

IV. Exhaustive path testing - test every combination of paths to find out if there are potential dependency problems.

First, note that I and II are NOT the same. For example, in EXAMPLE 1 above, you could have statement coverage without tracing either of the two branches 3-5 or 4-6.

Coverage II and III are also distinct. Edge coverage simply requires every edge to be taken. If the test at a branch point is complicated, there may be many distinct ways in which a particular branch may be chosen. For example, in the statement

```
if ( (a<b) && (c>d) )  s1;
```

there are three ways in which the test could be false - a>=b and c>d, a<b and c<=d, and a>=b and c<=d. Each of these cases could affect the succeeding statements in a different way. Boundary analysis and equivalence partitioning can be applied in type III coverage to verify the correct operation of all parts of the test condition. For the statement shown here we might have test cases:

```
     a=1, b=1, ...
a=1, b=2 ...
a=3, b=3 ....    etc.
```

Condition coverage can find some problems not found by edge coverage, such as off-by-one, etc.

Path coverage is the most complex and demanding of all. It should include condition coverage but also tests for all interactions between statements in various parts of the program For example, consider the code segment

```
if ( c1 ) s1 ;
   else s2;
 if ( c2 ) s3 ;
   else s4;
```

Condition coverage could be obtained by two test cases, one in which c1 and c2 were both true and one in which c1 and c2 were both false. However, the test case in which c1 is true and c2 is false gives s1 followed by s4, for which the program might have a problem. For example, s1 and s4 might both increment a counter that should only be incremented once.

In the case of loops, boundary analysis and equivalence partitioning also play a role, this time based on the number of executions of the loop. Of course, there is only one natural boundary; a loop can be executed zero times or more than zero times (or 1 time and more than one time in the case of a DO-WHILE loop). Note, in one sense in a program with a loop there are infinitely many paths. Normally we view the loop has having just two paths - the one from the branch point through the loop and back to the branch point, and the one from the branch point out of the loop. However, for full path coverage one would want to consider test cases that tried several combinations of number of times around the loop - zero times, exactly one time, 5 times, 20 times, etc.

A good strategy for structural testing is to develop test cases incrementally, working from the simplest coverage to the most complex. Starting with statement coverage can help uncover simple mistakes like a mistyped variable name or an error in an expression. Then edge coverage followed by full condition coverage can help verify that the basic branching and decisions are correct. Finally, path coverage test cases can help verify there are no unwanted interactions between statements in different parts of the program.

2.3 Final Comments

NEITHER STRUCTURAL NOR FUNCTIONAL TESTING IS ENOUGH BY ITSELF. For example, if you have forgotten a case, then structural testing won't help you find it because there is no corresponding statement/path in the program. On the other hand, a module's structural boundaries may not correspond to the problems boundaries, so functional testing may not exercise some path or test sufficiently.

HENSCHEN'S MAXIM - IN A WELL DESIGNED SOFTWARE PACKAGE, THE FUNCTIONAL AND STRUCTURAL BOUNDARIES AND PATHS SHOULD MATCH DOWN TO THE LEVEL WHERE THE PROBLEM SPECIFICATION LEAVES OFF.