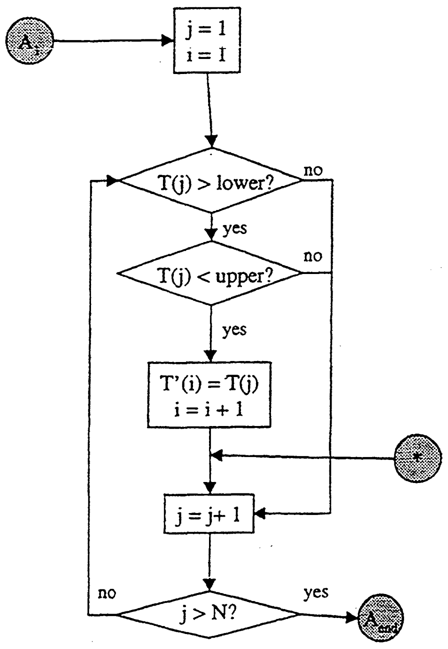
Chapter 8: Testing the programs

1. In the HP scheme, each fault has just one origin, one mode, and one or two types. For example, code errors have just one type, while design errors have two. The origins and modes are all orthogonal, and the type categories in each box in the diagram are orthogonal to the other categories in the same box. So a fault could have two types, from two different boxes, but would not fit into two type categories in the same box.

2. Assertions:

   A_1: (T is an array) & (size(T) = N) & (lower < upper)

   A_{end}: (T' is an array) & (size(T') $\leq$ N) & ($\forall$ i in 1..size(T'), T'(i) $\geq$ lower & T'(i) $\leq$ upper & $\exists$ j in 1..N, T(j) = T'(i))



3. First, the input assertion is:

   A_1: (T is an array) & (T is of size N)

   The assertion that is invariant for the inner loop (the one that goes through the array one element at a time) is the following. It is true at the beginning of every iteration of the loop (i.e. just before i is incremented):

   A_2: $i > 0 \,\&\, i < N \rightarrow [(T'(i) \leq T'(i+1)]$

   The assertion that is invariant for the outer loop (the one that keeps starting a new pass through the array) is the following. It is true at the beginning of every iteration of the loop (i.e. just before i is set to 0):

   A_3: $[not(more) = true)] \rightarrow [(T'$ sorted]

   Finally, the assertion that is true at the end of the program is the following:

   A_{end}: (T' is an array) & ($\forall$ i if $i < N$ then (T'(i) $\leq$ T(i+1))
   & ($\forall$ i if $i \leq N$ then $\exists\, j$ (T'(i) = T(j)) & (T' is of size N)

One possible path from the input condition to the output condition is:

$A_1 \rightarrow A_3 \rightarrow A_2 \rightarrow A_3 \rightarrow A_{end}$

The proof that these assertions hold on this path is outlined below:

- At the beginning of the first time through the outer loop, *more* has just been sent to *true*, so the antecedent of A_3 is false, thus A_3 holds.
- At the beginning of the first time through the inner loop, *i* has just been set to 0, so the antecedent of A_2 is false, thus A_2 holds.
- At the beginning of the second time through the outer loop, if the inner loop has only executed once, then $N = 1$ and so by definition *T* must be sorted, thus A_3 holds.
- At the end of the program, since $N = 1$, A_{end} must hold.

4. In general, a program with N 2-branch decisions would require 2^N test cases. A program with N M-branch decisions would require M^N test cases. These are in the worst case, when the decisions are sequential. The number of paths can be reduced by nesting the decisions. For example, the following program with 4 decisions has $2^4 = 16$ paths:

```
if (a < 0) and (b < 0) then ...
if (a < 0) and (b ≥ 0) then ...
if (a ≥ 0) and (b < 0) then ...
if (a ≥ 0) and (b ≥ 0) then ...
```

The following equivalent program has 3 decisions, but they are nested so that there are only 4 possible paths:

```
if (a < 0) then
        if (b < 0) then ...
        else ...
else
        if (b < 0) then ...
        else ...
```

5. Statement testing requires that the set of test cases exercises all the statements in the program. If each statement is a node in the graph, and each test case can be represented as a path through the graph, then statement testing would be equivalent to finding a set of paths that, together, cover all nodes. Branch testing must cover all branches of every decision, so in terms of the graph we must be concerned with edges not nodes. That is, in order to cover all branches of a particular decision, not only the node corresponding to that decision, but all the edges leading out of it must be covered. So branch testing is equivalent to finding a set of paths that cover all edges. Path testing must cover all branches and all statements and all possible combinations of branches and statements, so all paths in the graph must be covered.

6. One approach is to use a depth-first search from each node in the graph. The general algorithm would be:

```
for each node n in the graph do
  find_paths({n}, n)
```

where:

```
procedure find_paths (path, n)
begin
  for each node n2 such that (n, n2) is an edge in the graph do
        path = path + n2
        find_paths (path, n2)
  end
  if no such n2 exists then
        output path
```

This algorithm does not terminate if there are cycles in the graph, so a more complex algorithm would have to be used if that is a possibility. Otherwise, this algorithm works well even if there are many paths to be calculated because only one path is calculated at a time. However, it would become less efficient with graphs with very long paths because the depth of recursion would equal the length of the path being calculated. Thus, for graphs that tend to have fewer but longer paths, a breadth-first search design might be preferred.

7. Bottom-up:
1) Test E, G, H, J, K, L, M, and N
2) Test (F, L) and (I, M, N)
3) Test (B, F, L, G), (C, H), and (D, I, J, K, M, N)
4) Test (A..N)

Top-down:
1) Test A
2) Test (A, B, C, D, E)
3) Test (A, B, C, D, E, F, G, H, I, J, K)
4) Test (A..N)

Modified top-down:
1) Test A
2) Test B, C, D, and E
3) Test (A, B, C, D, E)
4) Test F, G, H, I, J, and K
5) Test (A, B, C, D, E, F, G, H, I, J, K)
6) Test L, M, and N
7) Test (A..N)

Big-bang:
1) Test A, B, C, D, E, F, G, H, I, J, K, L, M, and N
2) Test (A..N)

Sandwich:
1) Test A, L, M, and N
2) Test (A, B), (A, C), (A, D), (A, E), (F, L), and (I, M, N)
3) Test (B, F, G), (C, H), (D, I, J, K)
4) Test (A..N)

Modified Sandwich:
1) Test A, B, C, D, E, F, I, L, M, and N
2) Test G, H, J, K, (A, B), (A, C), (A, D), (A, E), (F, L), and (I, M, N)
3) Test (B, F, G), (C, H), (D, I, J, K)
4) Test (A..N)

8. The graph, and the research it is based on, indicate that, if many faults are found early (e.g. at compile time), there are likely to be many more faults still undetected. When that happens, one can assume that a lot of effort will be expended later in the lifecycle on finding and fixing those faults, and so it would be better to start the effort over again and write new code with fewer faults from the beginning.

9. It may be that large numbers of faults found early on indicate sloppy programming practices, which would result in even more faults found later. Or, maybe if there are many faults found early on, then a lot of changes will be made to the code to fix those faults, thus resulting in poorly structured code with lots more faults.

10. If N is the total number of indigenous faults in the program, then

$$N = \frac{\textit{(total number of seeded faults)} * \textit{(number of indigenous faults found)}}{\textit{number of seeded faults found}}$$

$$= \frac{25 * 5}{13}$$

$$= \textit{about } 10$$

So the number of indigenous faults remaining is $10 - 5 = 5$.

11. We will need to seed S faults, where:

$$\frac{S}{(S - 0 + 1)} = \frac{95}{100}$$

$$S = 19$$

The Richards formula requires that:

$$\frac{\binom{S}{s-1}}{\binom{S+1}{s}} = 0.95$$

Simplifying:

$$\frac{\binom{S}{s-1}}{\binom{S+1}{s}} = 0.95$$

$$\frac{\frac{S!}{(S-s+1)!\,(s-1)!}}{\frac{(S+1)!}{(S+1-s)!\,s!}} = 0.95$$

$$\frac{S!\,s!}{(s-1)!\,(S+1)!} = 0.95$$

$$\frac{s}{S+1} = 0.95$$

So the Richards formula gives the required ratio between total seeded and found seeded faults, not an absolute number for the number of faults which should be seeded.

12. In the testing of any system, there is a tradeoff between the thoroughness of the test and the resources available for testing. A more thorough test requires more resources. The differences in testing these three types of systems lie in how the tradeoff is resolved. In general, large amounts of resources are committed to testing a safety-critical system in order to ensure a very thorough test. The testing would also include checking for many types of error conditions, as well as many different combinations of input data, many of them unlikely to occur in practice. Fewer resources would be expended on the testing of a business-critical system, in order to apply those resources to other parts of the development process or to profit. A system not likely to affect lives, health or business would be tested with yet fewer resources. In addition, testing of these systems might very well concentrate more on user interface issues and other aspects that enhance marketability.

13. Consider a system that takes in sensor data from a large number of environmental sensors and performs calculations that combine data from different sensors. Different sensors have different patterns of data transmittal, and there are often unpredictable delays in receiving data. However, the calculations require that the data used were all recorded at the same moment, regardless of when they were received by the central system. Furthermore, certain calculations "go bad" in the sense that, if they cannot be calculated correctly within a certain amount of time, they are no longer useful and processing must proceed to the next calculation. In the design of the system, each sensor is treated as an object, as well as each module that performs a set of related computations. Some computational objects also pass the results of their computations to other modules as input to other time-sensitive calculations, further complicating the timing complexity. Testing such a system would require testing a large variety of different timing sequences, including data which arrive too late to be useful, data which arrive in an unpredictable order, and propagation of calculated data.

14. Some issues to consider:

- How was responsibility for failures handled in the contract between the developing organization and the independent test team, if there was a contract?

- What measures did the development organization take to ensure that testing was carried out properly?
- Who was responsible for the test plan?

15. The strategy and test plan could include:
    - For each of the three components, a unit test plan which outlines test cases for:
        - each function implemented by the component
        - valid and invalid input data or user commands
        - various user scenarios
    - Plans for any inspections or reviews that will take place
    - An integration plan which outlines:
        - the order in which components will be integrated
        - how the interfaces between components will be tested
        - the role, if any, of an independent testing team
    - A system test plan which includes
        - usability testing
        - the customer's role in this phase of testing
    - Deadlines or targets for all testing activities
    - Criteria for determining when testing is complete