BONUS,LEVEL and HITS and returns a (possibly new) value for BONUS.It might be written thus:

```
6030 IF LEVEL>2 AND HITS=10 THEN
   BONUS=BONUS*LEVEL
6040 IF LEVEL=6 OR BONUS>2000 THEN
   BONUS=BONUS+100
```

To cover the outcome of each conditional expression, we need to consider the inputs to each that would cause an output of 'yes' or 'no'. In both decisions we are looking at the effects of two variables combined by a logical operator (AND and OR). This means that we have to take the combined values of the variables and not their individual values into consideration. To see why, consider what would happen if we tested values for LEVEL of 4 and 1 and for HITS of 10, 5 and 20 in the first decision. When LEVEL=4, the three values of HITS are tested but when LEVEL=1 they are not. This is a case of part of a decision 'masking' another part. So that we can test each part separately, it is best to simplify compound decisions.

Looking at figure 3, we can see that with four binary decisions there are 2^4 (=16) possible outcomes and we must cover them all. A start is to list the conditions for a yes or no outcome for each decision like this:

	1	2	3	4
yes	LEVEL>2	HITS=10	LEVEL=6	BONUS>20000
no	LEVEL=2 LEVEL<2	HITS<10 HITS>10	LEVEL<6 LEVEL>6	BONUS=2000 BONUS<2000

These can then be used to derive the values for representative test data. For instance, for the path taking the route adfi (see figure 3), LEVEL must be greater than 2 and equal to 6, HITS must be not equal to 10 and BONUS may be any value (because it is not involved). The values LEVEL=6, HITS=20 and BONUS=150 would exercise this path — as would many others, of course. The route abehj could be tested with LEVEL=4, HITS=10 and BONUS=600 (don't forget we are talking about the *input* value of BONUS that may later be multiplied with LEVEL).

Equally importantly, the results that should be produced by each set of test data should be calculated before the test run so that the results can be compared. The input data on their own will merely test whether the program runs. To test that it is doing what it should, the output must be calculated (by hand) beforehand. A complete set of test cases for this example is shown (left).

Equipped with a method of 'exercising' our software, we now need a way of tackling a large program so that the complexity does not become overwhelming. It is here that another benefit of structured programming is felt. Programs written as a collection of independent modules arranged in a hierarchy allow us to test each module individually. Because the modules are arranged in this way, we can start with the topmost module and work down, testing each individual module only when all of those above it have been tested, and we can use already-tested modules to provide data for those lower in the structure.

The module being tested will have above it (unless it is the first one), a fully tested *driver* module. The modules below it, known as *stubs*, are, so far, untested and therefore unreliable, so they are simulated by short pieces of code that simply return the appropriate test data when called by the module being tested. This arrangement is sometimes known as a *test harness* and it is a framework into which module routines can be put for testing. Figure 4 shows the principle. Modules 1, 2 and 3 have already been tested while modules 5, 6 and 7 are simulated.

One final point must be stressed. Testing is an important part of the program's life cycle and, as such, deserves to be well documented. It pays to keep records of the test data derived for a routine so that, if it shows a bug later, the same tests will not have to be repeated, or the testing can be examined for where it was inadequate.

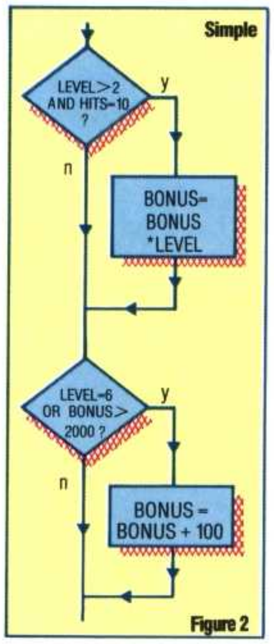

Figure 2

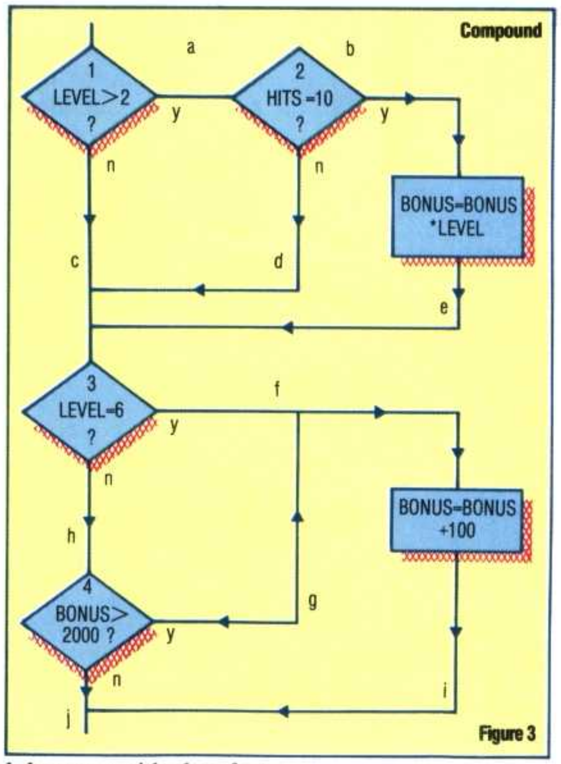

Figure 3

Decision Masking
Simplifying compound decisions and labelling the flowchart links makes systematic testing easier

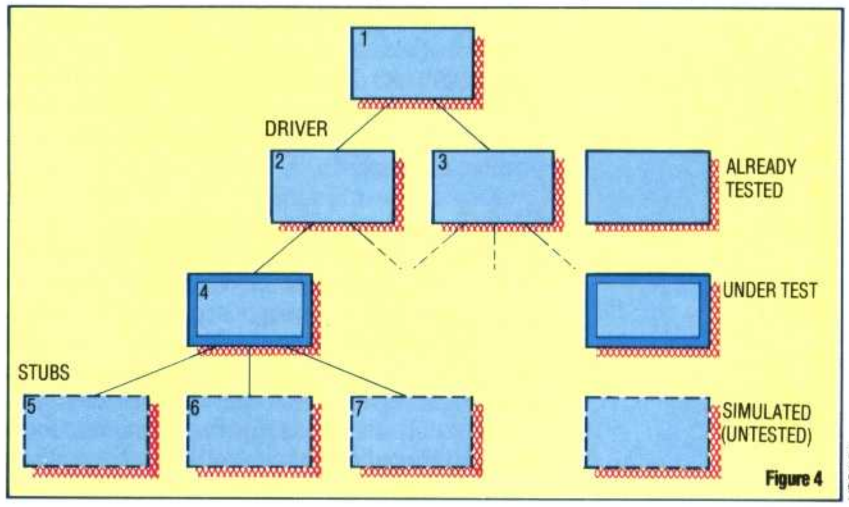

Figure 4

Top-Down Testing
Testing is made much simpler by the top-down approach, since each module can be tested as it is written, both in isolation and in association with other tested modules. The behaviour of unwritten modules can be simulated by writing 'stubs' — code that artifically generates examples of the module's predicted output

LIZ DIXON