8.2 UNIT TESTING

Before we discuss the intricacies of unit testing, we shall first highlight some general concepts involved in testing.

8.2.1 What is Testing?

Testing a program consists of providing the program with a set of test inputs (or test cases) and observing if the program behaves as expected. If the program fails to behave as expected, then the conditions under which a failure occurs are noted for debugging and correction. The following are some commonly used terms associated with testing:

- A **failure** is a manifestation of an **error** (or *defect* or *bug*). But, the mere presence of an error may not necessarily lead to a failure.
- A **fault** is an incorrect intermediate state that may have been entered during program execution, e.g. a variable value is different from what it should be. A fault may or may not not lead to a failure.
- A **test case** is the triplet [I, S, O], where I is the data input to the system, S is the state of the system at which the data is input, and O is the expected output of the system.
- A **test suite** is the set of all test cases with which a given software product is to be tested.

8.2.2 Verification vs. Validation

Verification is the process of determining whether one phase of a software product conforms to its previous phase, whereas validation is the process of determining whether a fully developed system

conforms to its requirements specification. Thus while verification is concerned with phase containment of errors, the aim of validation is to make the final product error free.

8.2.3 Design of Test Cases

Exhaustive testing of almost any non-trivial system is impractical due to the fact that the domain of input data values to most practical software systems is either extremely large or infinite. Therefore, we must design an *optimal test suite* that is of reasonable size and can uncover as many errors existing in the system as possible. Actually, if test cases are selected randomly, many of them do not contribute to the *significance* of the test suite, i.e. they do not detect any additional errors not already being detected by other test cases in the suite. Thus, the number of random test cases in a test suite is, in general, not an indication of the effectiveness of the testing. In other words, testing a system using a large collection of test cases that are selected at random does not guarantee that all (or even most) of the errors in the system will be uncovered. Consider the following example code segment for finding the maximum of two integer values x and y. This code segment has a simple programming error.

```
If (x>y) max = x;
else max = x;
```

For the above code segment, the test set {(x = 3, y = 2); (x = 2, y = 3)} can detect the error, whereas a larger test set {(x = 3, y = 2); (x = 4, y = 3); (x = 5, y = 1)} does not detect the error.

Thus, systematic approaches are required to design optimal test sets in which each test case is designed to detect different errors. There are essentially two main approaches to designing test cases:

- Black-box approach
- White-box (or glass-box) approach

In the black-box approach, test cases are designed using only the functional specification of the software, i.e. without any knowledge of the internal structure of the software. For this reason, black-box testing is also known as *functional testing*. On the other hand, designing white-box test cases requires thorough knowledge of the internal structure of software, and therefore the white-box testing is also called the *structural testing*.

8.2.4 Driver and Stub Modules

In order to test a single module, we need a complete environment to provide all that is necessary for execution of the module. That is,

besides the module under test itself, we need the following in order to be able to test the module:

- The procedures the module under test calls and which do not belong to it.
- Nonlocal data structures that the module accesses.
- A procedure to call the functions of the module under test with appropriate parameters.

Thus, in order to test a single module, we have to provide a complete environment for its execution. However, since the required modules (which either call or are called by the module under test) are usually not available until they too have been unit tested, *stubs* and *drivers* are designed to provide the complete environment for a module. The role of stub and driver modules is shown in Fig. 8.1. A stub procedure for a given procedure is a dummy procedure that has the same I/O parameters as the given procedure but has a highly

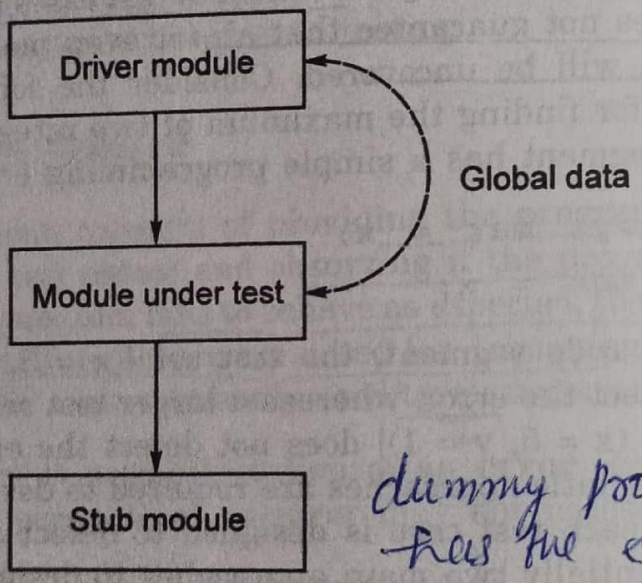

Fig. 8.1 Unit testing with the help of driver and stub modules.

simplified behaviour. For example, a stub procedure may produce the expected behaviour using table lookup. A driver module would contain the nonlocal data structures accessed by the module under test, and would also have the code to call the different functions of the module with appropriate parameter values.

8.3 BLACK-BOX TESTING

There are essentially the following two main approaches to designing black-box test cases.

- Equivalence class partitioning
- Boundary value analysis

8.3.1 Equivalence Class Partitioning

In this approach, the domain of input values to a program is partitioned into a set of equivalence classes. This partitioning is done such that the behaviour of the program is similar for every input data belonging to the same equivalence class. The main idea of defining the equivalence classes is that testing the code with any one value belonging to an equivalence class is as good as testing the software with any other value belonging to that equivalence class. Equivalence classes for a software can be designed by examining the input data. The following are some general guidelines for designing the equivalence classes:

- If the input data values to a system can be specified by a range of values, then one valid and two invalid equivalence classes should be defined.

- If the input can assume values from a set of discrete members of some domain, then one equivalence class for valid input values and another equivalence class for invalid input values should be defined.

Example 1. For a software that computes the square root of an input integer which can assume values in the range from 1 to 5000, there are three equivalence classes: the set of negative integers, the set of integers in the range from 1 to 5000, and the integers larger than 5000. Therefore, the test cases must include representatives from each of the three equivalence classes and a possible test set can be: {– 5, 500, 6000}.

8.3.2 Boundary Value Analysis

Some typical programming errors occur at the boundaries of different equivalence classes of inputs. The reason for such errors might purely be due to psychological factors. Programmers often fail to see the special processing required by the input values that lie at the boundary of the different equivalence classes. For example, programmers may improperly use < instead of <=, or conversely. Boundary value analysis leads to selection of test cases at the boundaries of the different equivalence classes.

Example 2. For a function that computes the square root of integer values in the range from 1 to 5000, the test cases must include the following values: {0, 1, 5000, 5001}.

8.4 WHITE-BOX TESTING

There are several methodologies used for white-box testing. We discuss some important ones below.