
SOFTWARE TESTING TECHNIQUES

Testing Objectives:

In an excellent book on software testing, Glen Myers states a number of rules that can serve well as testing objectives:

1. Testing is a process of executing a program with the intent of finding an error.
2. A good test case is one that has a high probability of finding an as-yet undiscovered error.
3. A successful test is one that uncovers an as-yet-undiscovered error.

Testing Principles:

Before applying methods to design effective test cases, a software engineer must understand the basic principles that guide software testing. Davis suggests a set of testing principles:

- All tests should be traceable to customer requirements. As we have seen, the objective of software testing is to uncover errors. It follows that the most severe defects (from the customer's point of view) are those that cause the program to fail to meet its requirements.
- Tests should be planned long before testing begins. Test planning can begin as soon as the requirements model is complete.

Detailed definition of test cases can begin as soon as the design model has been solidified. Therefore, all tests can be planned and designed before any code has been generated.

- Testing should begin “in the small” and progress toward testing “in the large.” The first tests planned and executed generally focus on individual

components. As testing progresses, focus shifts in an attempt to find errors in integrated clusters of components and ultimately in the entire system.

- Exhaustive testing is not possible. The number of path permutations for even a moderately sized program is exceptionally large. For this reason, it is impossible to execute every combination of paths during testing. It is possible, however, to adequately cover program logic and to ensure that all conditions in the component-level design have been exercised.
- To be most effective, testing should be conducted by an independent third party. By most effective, we mean testing that has the highest probability of finding errors (the primary objective of testing). The software engineer who created the system is not the best person to conduct all tests for the software.

Test Case Design:

The design of tests for software and other engineered products can be as challenging as the initial design of the product itself. A rich variety of test case design methods have evolved for software. These methods provide the developer with a systematic approach to testing. More important, methods provide a mechanism that can help to ensure the completeness of tests and provide the highest likelihood for uncovering errors in software.

Any engineered product (and most other things) can be tested in one of two ways:

- (1) Knowing the specified function that a product has been designed to perform, tests can be conducted that demonstrate each function is fully operational while at the same time searching for errors in each function;
- (2) knowing the internal workings of a product, tests can be conducted to ensure that "all gears mesh," that is, internal operations are performed

according to specifications and all internal components have been adequately exercised. The first test approach is called ***black-box testing*** and the second, ***white-box testing***.

When computer software is considered, black-box testing alludes to tests that are conducted at the software interface. Although they are designed to uncover errors, black-box tests are used to demonstrate that software functions are operational, that input is properly accepted and output is correctly produced, and that the integrity of external information (e.g., a database) is maintained. A black-box test examines some fundamental aspect of a system with little regard for the internal logical structure of the software.

White-box testing of software is predicated on close examination of procedural detail. Logical paths through the software are tested by providing test cases that exercise specific sets of conditions and/or loops. The "status of the program" may be examined at various points to determine if the expected or asserted status corresponds to the actual status.

At first glance it would seem that very thorough white-box testing would lead to "100 percent correct programs." All we need do is define all logical paths, develop test cases to exercise them, and evaluate results, that is, generate test cases to exercise program logic exhaustively. Unfortunately, exhaustive testing presents certain logistical problems. For even small programs, the number of possible logical paths can be very large. For example, consider the 100 line program in the language C. After some basic data declaration, the program contains two nested loops that execute from 1 to 20 times each, depending on conditions specified at input. Inside the interior loop, four if-then-else constructs are required. There are approximately 10^{14} possible paths that may be executed in this program!

To put this number in perspective, we assume that a magic test processor has been developed for exhaustive testing. The processor can develop a test case, execute it, and evaluate the results in one millisecond.

Working 24 hours a day, 365 days a year, the processor would work for 3170 years to test the program. Exhaustive testing is impossible for large software systems.

White-box testing should not, however, be dismissed as impractical. A limited number of important logical paths can be selected and exercised. Important data structures can be probed for validity. The attributes of both black- and white-box testing can be combined to provide an approach that validates the software interface and selectively ensures that the internal workings of the software are correct.

WHITE-BOX TESTING

White-box testing, sometimes called glass-box testing, is a test case design method that uses the control structure of the procedural design to derive test cases. Using white-box testing methods, the software engineer can derive test cases that:

- (1) guarantee that all independent paths within a module have been exercised at least once,
- (2) exercise all logical decisions on their true and false sides,
- (3) execute all loops at their boundaries and within their operational bounds,
- and
- (4) exercise internal data structures to ensure their validity.