Table of Contents

Introduction

Because of the fallibility of its human designers and its own abstract, complex nature, software development must be accompanied by quality assurance activities. It is not unusual for developers to spend 40% of the total project time on testing. For life-critical software (e.g. flight control, reactor monitoring), testing can cost 3 to 5 times as much as all other activities combined. The *destructive* nature of testing requires that the developer discard preconceived notions of the *correctness* of his/her developed software.

...(BACK TO TOP)

Software Testing Fundamentals

Testing objectives include

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 should systematically uncover different classes of errors in a minimum amount of time and with a minimum amount of effort. A secondary benefit of testing is that it demonstrates that the software appears to be working as stated in the specifications. The data collected through testing can also provide an indication of the software's reliability and quality. But, testing cannot show the absence of defect -- it can only show that software defects are present.

...(BACK TO TOP)

White Box Testing

White box testing is a test case design method that uses the control structure of the procedural design to derive test cases. Test cases can be derived 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.

...(BACK TO TOP)

The Nature of Software Defects

Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will be executed. General processing tends to be well understood while *special case* processing tends to be prone to errors.

We often believe that a logical path is not likely to be executed when it may be executed on a regular basis. Our unconscious assumptions about control flow and data lead to design errors that can only be detected by path testing.

Typographical errors are random.

...(BACK TO TOP)

Basis Path Testing

This method enables the designer to derive a logical complexity measure of a procedural design and use it as a guide for defining a *basis set* of execution paths. Test cases that exercise the basis set are guaranteed to execute every statement in the program at least once during testing.

Flow Graphs

Flow graphs can be used to represent control flow in a program and can help in the derivation of the basis set. Each flow graph *node* represents one or more procedural statements. The *edges* between nodes represent flow of control. An edge must terminate at a node, even if the node does not represent any useful procedural statements. A *region* in a flow graph is an area bounded by edges and nodes. Each node that contains a *condition* is called a *predicate node*. *Cyclomatic complexity* is a metric that provides a quantitative measure of the logical complexity of a program. It defines the number of *independent paths* in the basis set and thus provides an upper bound for the number of tests that must be performed.

...(BACK TO TOP)

The Basis Set

An independent path is any path through a program that introduces at least one new set of processing statements (must move along at least one new edge in the path). The basis set is not *unique*. Any number of different basis sets can be derived for a given procedural design. Cyclomatic complexity, $V(G)$, for a flow graph G is equal to

1. The number of regions in the flow graph.
2. $V(G) = E - N + 2$ where E is the number of edges and N is the number of nodes.
3. $V(G) = P + 1$ where P is the number of predicate nodes.

...(BACK TO TOP)

Deriving Test Cases

1. From the design or source code, derive a flow graph.
2. Determine the cyclomatic complexity of this flow graph.
    - Even without a flow graph, $V(G)$ can be determined by counting the number of conditional statements in the code.
3. Determine a basis set of linearly independent paths.
    - Predicate nodes are useful for determining the necessary paths.

4. Prepare test cases that will force execution of each path in the basis set.
    - Each test case is executed and compared to the expected results.

…**(BACK TO TOP)**

Automating Basis Set Derivation

The derivation of the flow graph and the set of basis paths is amenable to automation. A software tool to do this can be developed using a data structure called a *graph matrix*. A graph matrix is a square matrix whose size is equivalent to the number of nodes in the flow graph. Each row and column correspond to a particular node and the matrix corresponds to the connections (edges) between nodes. By adding a *link weight* to each matrix entry, more information about the control flow can be captured. In its simplest form, the link weight is 1 if an edge exists and 0 if it does not. But other types of link weights can be represented:

- the probability that an edge will be executed,
- the processing time expended during link traversal,
- the memory required during link traversal, or
- the resources required during link traversal.

Graph theory algorithms can be applied to these graph matrices to help in the analysis necessary to produce the basis set.

…(BACK TO TOP)

Loop Testing

This white box technique focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined:

1. simple loops,
2. nested loops,
3. concatenated loops, and
4. unstructured loops.

Simple Loops

The following tests should be applied to simple loops where *n* is the maximum number of allowable passes through the loop:

1. skip the loop entirely,
2. only pass once through the loop,

3. *m* passes through the loop where $m < n$,
4. *n - 1, n, n + 1* passes through the loop.

Nested Loops

The testing of nested loops cannot simply extend the technique of simple loops since this would result in a geometrically increasing number of test cases. One approach for nested loops:

1. Start at the innermost loop. Set all other loops to minimum values.
2. Conduct simple loop tests for the innermost loop while holding the outer loops at their minimums. Add tests for out-of-range or excluded values.
3. Work outward, conducting tests for the next loop while keeping all other outer loops at minimums and other nested loops to *typical* values.
4. Continue until all loops have been tested.

Concatenated Loops

Concatenated loops can be tested as simple loops if each loop is *independent* of the others. If they are not independent (e.g. the loop counter for one is the loop counter for the other), then the nested approach can be used.

Unstructured Loops

This type of loop should be **redesigned** not tested!!!

…(BACK TO TOP)

Other White Box Techniques

Other white box testing techniques include:

1. Condition testing
    - exercises the logical conditions in a program.
2. Data flow testing
    - selects test paths according to the locations of definitions and uses of variables in the program.

…(BACK TO TOP)

Black Box Testing

Introduction

Black box testing attempts to derive sets of inputs that will fully exercise all the *functional requirements* of a system. It is **not** an alternative to white box testing. This type of testing attempts to find errors in the following categories:

1. incorrect or missing functions,
2. interface errors,
3. errors in data structures or external database access,
4. performance errors, and
5. initialization and termination errors.

Tests are designed to answer the following questions:

1. How is the function's validity tested?
2. What classes of input will make good test cases?
3. Is the system particularly sensitive to certain input values?
4. How are the boundaries of a data class isolated?
5. What data rates and data volume can the system tolerate?
6. What effect will specific combinations of data have on system operation?

White box testing should be performed early in the testing process, while black box testing tends to be applied during later stages. Test cases should be derived which

1. reduce the number of additional test cases that must be designed to achieve reasonable testing, and
2. tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

…**(BACK TO TOP)**

Equivalence Partitioning

This method divides the input domain of a program into classes of data from which test cases can be derived. Equivalence partitioning strives to define a test case that uncovers classes of errors and thereby reduces the number of test cases needed. It is based on an evaluation of equivalence classes for an *input condition*. An *equivalence class* represents a set of valid or invalid states for input conditions.

Equivalence classes may be defined according to the following guidelines:

1. If an input condition specifies a *range*, one valid and two invalid equivalence classes are defined.
2. If an input condition requires a specific *value*, then one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a *set*, then one valid and one invalid equivalence class are defined.
4. If an input condition is *boolean*, then one valid and one invalid equivalence class are defined.

...**(BACK TO TOP)**

Boundary Value Analysis

This method leads to a selection of test cases that exercise boundary values. It complements equivalence partitioning since it selects test cases at the edges of a class. Rather than focusing on input conditions solely, BVA derives test cases from the *output domain* also. BVA *guidelines* include:

1. For input *ranges* bounded by *a* and *b*, test cases should include values *a* and *b* and just above and just below *a* and *b* respectively.
2. If an input condition specifies a number of values, test cases should be developed to exercise the minimum and maximum numbers and values just above and below these limits.
3. Apply guidelines 1 and 2 to the output.
4. If internal data structures have prescribed boundaries, a test case should be designed to exercise the data structure at its boundary.

...**(BACK TO TOP)**

Cause-Effect Graphing Techniques

Cause-effect graphing is a technique that provides a concise representation of logical conditions and corresponding actions. There are four steps:

1. *Causes* (input conditions) and *effects* (actions) are listed for a module and an identifier is assigned to each.
2. A cause-effect graph is developed.
3. The graph is converted to a decision table.
4. Decision table rules are converted to test cases.

Author: D.A. Stacey
Date of Last Update: Wednesday, January 13, 1999 06:10:23 PM