Software Testing Techniques and Strategies (Chapter 8)

What is it?
Once source code has been generated, software must be tested to uncover (and correct) as many errors as possible before delivery to your customer.
Software testing techniques provide systematic guidance for designing tests that
(1) Exercise the internal logic of software components, and
(2) Exercise the input and output domains of the program to uncover errors in program function, behavior and performance.
Who does it?
During early stages of testing, a software engineer performs all tests.
However, as the testing process progresses, testing specialists may become involved.
Why is it important?
Reviews and other SQA activities can and do uncover errors, but they are not sufficient.
In order to find the highest possible number of errors, tests must be conducted systematically and test cases must be designed using disciplined techniques.
What are the steps?
Software is tested from two different perspectives:
1. Internal program logic is exercised using “white box” test case design techniques.
2. Software requirements are exercised using “black box” test case design techniques.

Testing is the process of exercising the program with the specific intent of finding the errors prior to delivery to the end user.

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

- **All tests should be traceable to customer requirements**
- **Tests should be planned long before testing begins.**
  Test planning can begin as soon as the requirements model is complete. All tests can be planned and designed before any code has been generated.
- **The Pareto principle applies to software testing.**
  It states that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components.
- **Testing should begin “in the small” and progress toward testing “in the large.”**
  The first tests planned and executed generally focuses on individual (small) components. As testing progresses, focus shifts in an attempt to find errors in the integrated components and ultimately in the entire.
- **Exhaustive testing is not possible.**
  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 design have been exercised.
- **To be most effective, testing should be conducted by an independent third party.**

- **Testability**

Software testability is simply how easily a computer program can be tested.

Characteristics (OOSSDUC)

Operability "The better it works, the more efficiently it can be tested."

Observability "What you see is what you test."

Simplicity "The less there is to test, the more quickly we can test it."

Stability "The fewer the changes, the fewer the disruptions to testing."

Decomposability "By controlling the scope of testing, we can more quickly isolate the errors and conduct smart testing.

Understandability "The more information we have, the smarter we will test."

Controllability "The better we can control the software, the more the testing can be optimized"

Good Test Case

1. A good test has a high probability of finding an error.
2. A good test is not redundant
3. A good test should be "best of breed"
4. A good test should be neither too simple nor too complex.

TEST CASE DESIGN STRATEGIES

- Black-box or behavioral testing (knowing the specified function a product is to perform and demonstrating correct operation based solely on its specification without regard for its internal logic)
- White-box or glass-box testing (knowing the internal workings of a product, tests are performed to check the workings of all independent logic paths)

White-box testing, sometimes called *glass-box testing* or *code testing,* is a test case design method that uses the control structure of the procedural design to derive test cases. (focus is on data flow, control flow ,information flow, coding practice , exception and error handling)
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.

Even though white box testing seems to be an ideal method for testing, it does not guarantee failures from faulty data. Secondly it is difficult to conduct such extensive and exhaustive testing in large organizations.

I. BASIS PATH TESTING

The basis path method enables the test case designer to derive a logical complexity measure of a procedural design. Now he uses this measure as a guide for deriving a basis set of execution paths. Test cases derived to exercise this basis set are guaranteed to execute every statement in the program at least one time during testing.

FLOW GRAPH

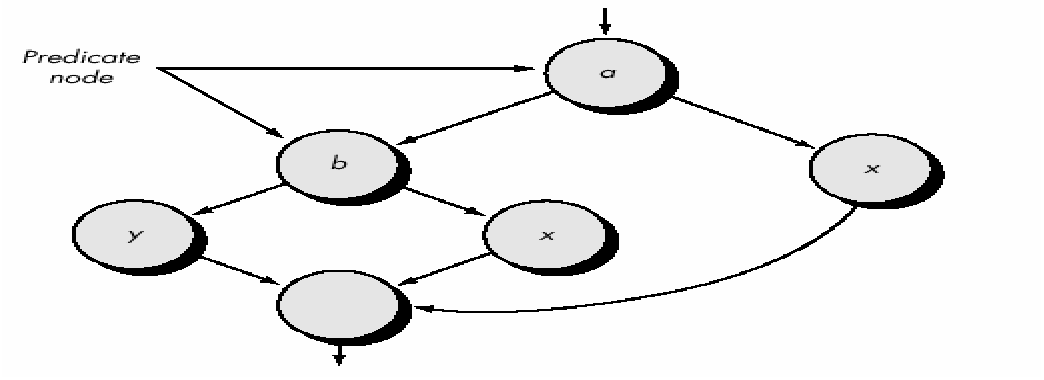
It is a simple notation for the depiction of the control flow.

Node – Each circle in the flow graph represents one or more procedural statements.

Edge – They represent the flow of control. An edge must always terminate on a node.

Region – An area bounded by edges and nodes is called a region. The area outside the graph is also considered as a region.

Predicate Node – each node that contains a condition is called a predicate node and is characterized by two or more no of branches emerging from it.



CYCLOMATIC COMPLEXITY

It is software metric which provides a quantitative measure of the logical complexity of the program. It defines the no of paths in the basis set of the program, hence establishes the no of tests that must be conducted to ensure that every statement is executed atleast once.

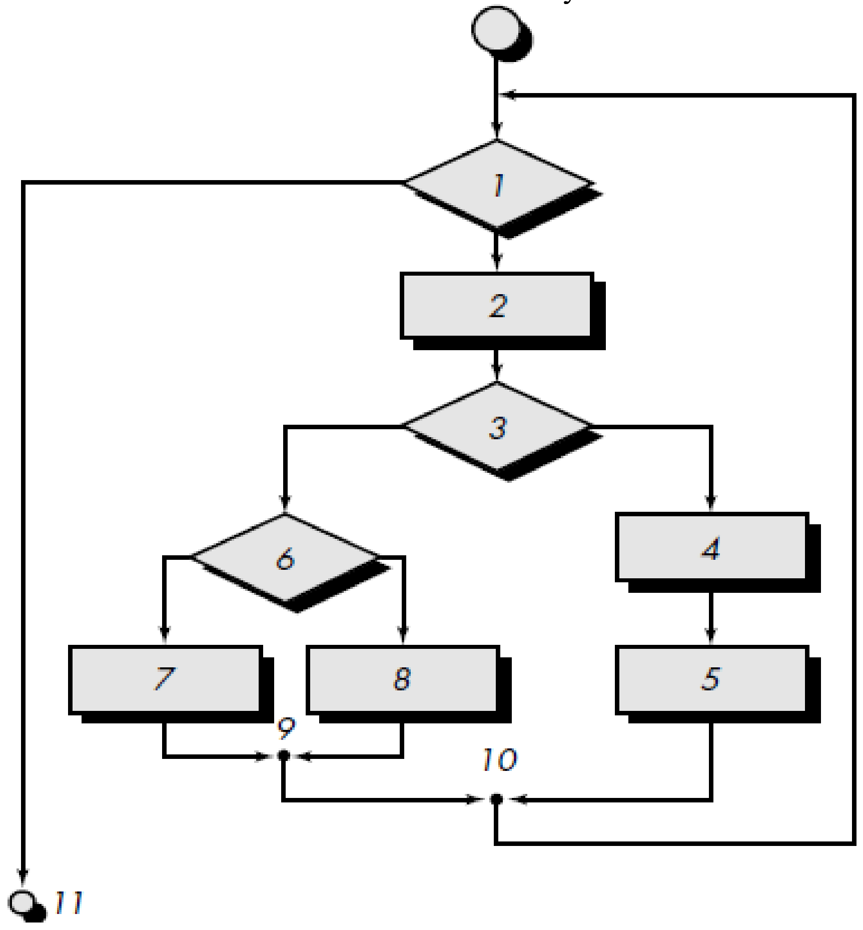


fig: Flow Chart

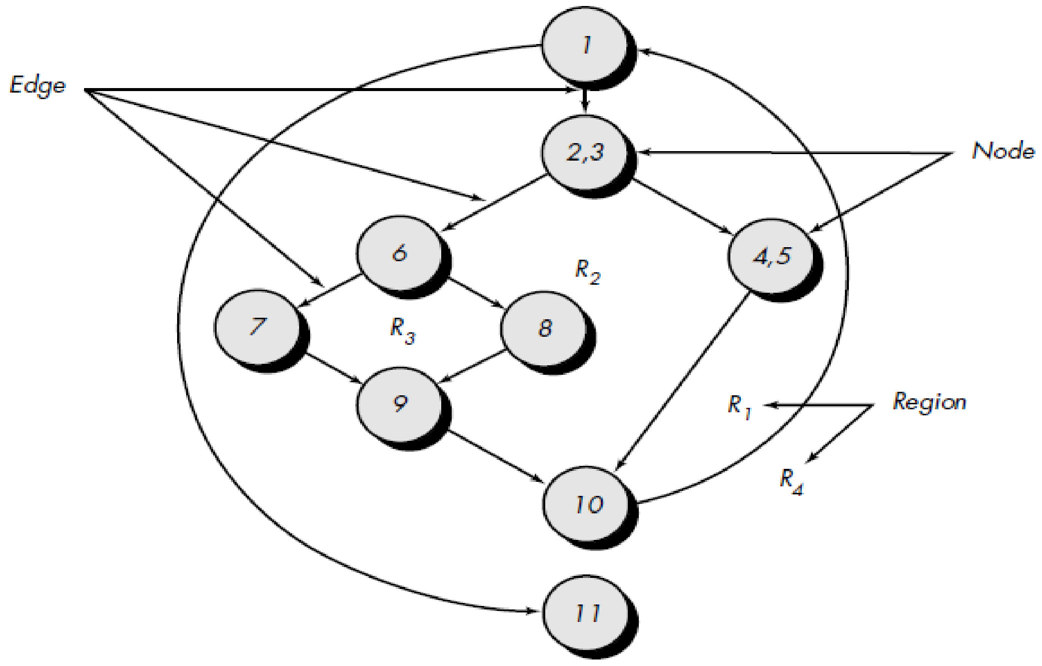


Fig: Flow Graph

Independent Path – any path in the program that introduces at least one new processing statement or condition. In terms of flow graph an independent path is the one that introduces atleast one edge which has not been traversed before.
path 1: 1-11
path 2: 1-2-3-4-5-10-1-11
path 3: 1-2-3-6-8-9-10-1-11
path 4: 1-2-3-6-7-9-10-1-11
Note that each new path introduces a new edge. The path
1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.

Complexity is computed in one of three ways:
1. The number of regions of the flow graph correspond to the cyclomatic complexity. From the above figure **The flow graph has four regions.**
2. Cyclomatic complexity, V(G), for a flow graph, *G,* is defined as
V(G) = E - N + 2
where E is the number of flow graph edges, N is the number of flow graph nodes.
From the Figure : **V(G) = 11 edges - 9 nodes + 2 = 4.**
3. Cyclomatic complexity, V(G), for a flow graph, *G,* is also defined as
V(G) = P + 1
Where P is the number of predicate nodes contained in the flow graph G.
From the Figure : **V(G) = 3 predicate nodes + 1 = 4.**

Hence cyclomatic complexity for the flow graph is 4.

II. CONTROL STRUCTURE TESTING

 The basis path testing technique is one of a number of techniques for white box testing.
 Although basis path testing is simple and highly effective, it is not sufficient in itself.
In this section, other variations on control structure testing are discussed.
 These broaden testing coverage and improve quality of white-box testing.

Condition Testing

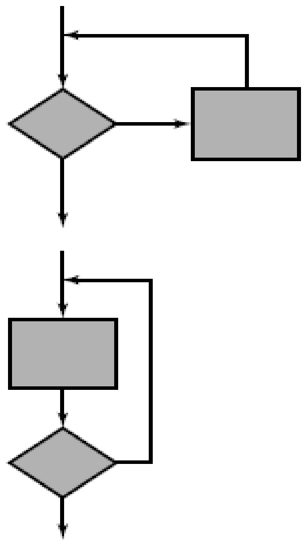
Condition testing is a test case design method that exercises the logical conditions contained in a program module.
If a condition is incorrect, then at least one component of the condition is incorrect.
E1 <relational-operator> E2
where E1 and E2 are arithmetic expressions and <relational-operator> is one of the following:
<, ≤, =, ≠ (nonequality), >, or ≥.
Therefore, types of errors in a condition include the following:
 Boolean operator error (incorrect/missing/extra Boolean operators).
 Boolean variable error.
 Boolean parenthesis error.
 Relational operator error.
 Arithmetic expression error.
The condition testing method focuses on testing each condition in the program. The purpose of condition testing is to detect not only errors in the conditions of a program but also other errors in the program.

Data Flow Testing

The *data flow testing* method selects test paths of a program according to the locations of definitions and uses of variables in the program.

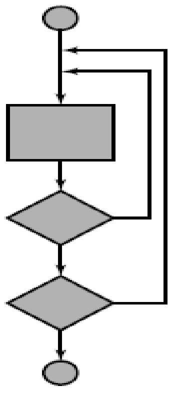
III. Loop Testing

Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs. Four different classes of loops can be defined: simple loops, concatenated loops, nested loops, and unstructured loops.
Simple loops: The following set of tests can be applied to simple loops, where *n* is the maximum number of allowable passes through the loop.
1. Skip the loop entirely.
2. Only one pass through the loop.
3. Two passes through the loop.
4. *m* passes through the loop where *m* < *n*.
5. *n* _1, *n*, *n* + 1 passes through the loop.


Simple loops

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 minimum iteration parameter (e.g., loop counter) values. Add other tests for out-of range or excluded values.
3. Work outward, conducting tests for the next loop, but keeping all other outer loops at minimum values and other nested loops to "typical" values.


Nested loops

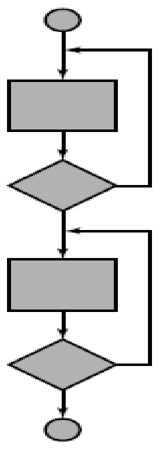
4. Continue until all loops have been tested.

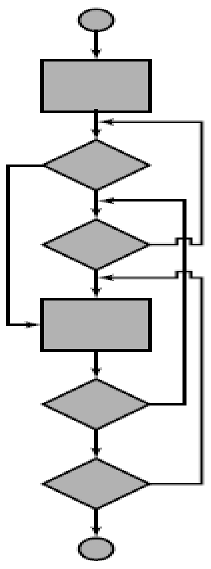
Concatenated loops:

Concatenated loops can be tested using the approach defined for simple loops, if each of the loops is independent of the other.
However, if two loops are concatenated and the loop counter for loop 1 is used as the initial value for loop 2, then the loops are not independent.
When the loops are not independent, the approach applied to nested loops is recommended.

Unstructured loops.

Whenever possible, this class of loops should be redesigned to reflect the use of the structured programming constructs.


Concatenated loops


Unstructured loops

BLACK-BOX TESTING

In this method the analyst examines the specifications to see what the program must do under various conditions. Test cases are then developed for a condition or a combination of conditions and submitted for processing. By examining the results the analyst can determine if the specified requirements are satisfied.

Black-box testing attempts to find errors in the following categories:

1. Incorrect or missing functions,
2. Interface errors,
3. Errors in data structures or external data base access,
4. Behavior or performance errors, and
5. Initialization and termination errors.

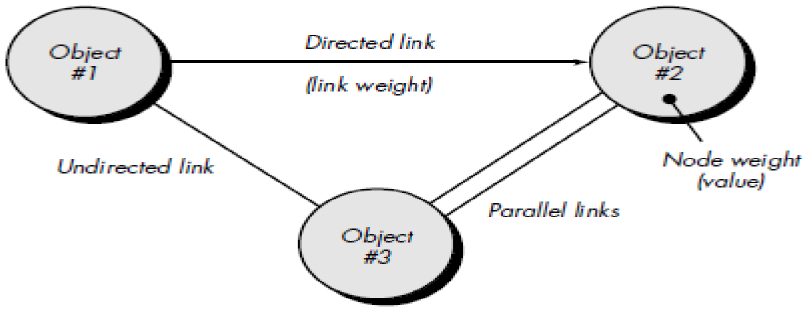
<u>TYPES OF BLACK BOX TESTING</u>

1. Graph-Based Testing Methods:

The first step in black-box testing is to understand the objects (nodes) that are modeled in software and the relationships (links) that connect these objects.

Once this has been accomplished, the next step is to define a series of tests that verify "all objects have the expected relationship to one another. Nodes are represented by circles and are connected via links. Types of Links –

Unidirectional
Bidirectional
Parallel (more than one relationship between two nodes)



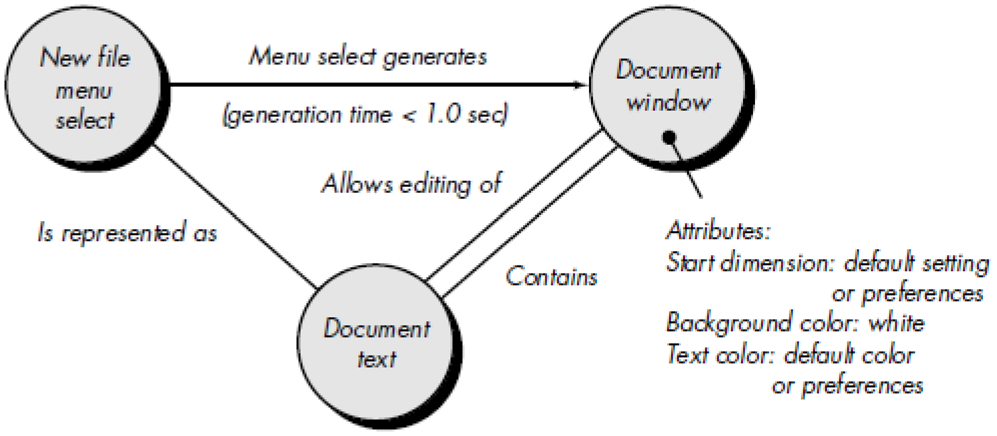


fig: An example for a word processing application

(a) Transaction Flow Modeling

Nodes represent the steps in the transactions & links represent logical connections.
– Data Flow Diagrams

(b) Finite State Modeling

Nodes represent the user observable states & links represents the transitions.
- State diagrams

(c) Data flow Modeling

Nodes represent the data objects & links represent the transformations from one data object to another.

(d) Timing Modeling

Nodes represent the objects and links represent the sequential connections between the objects. link weights are required execution times.

2. Equivalence Partitioning

Divide the input domain of the program into classes of data.
Derive test cases from these classes.
Such a test case would single handedly uncover a class of errors (say incorrect processing of all the character data).
Thus equivalence portioning strives to uncover classes of errors, thereby reducing the no of test cases to be developed.
If a set of objects are linked by relationships that are transitive, reflexive and symmetric, then equivalence is present.

An equivalence class represents a set of valid or invalid states.

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,* one valid and two invalid equivalence classes are defined.
3. If an input condition specifies a member of a *set,* one valid and one invalid equivalence class are defined.
4. If an input condition is *Boolean,* one valid and one invalid class are defined.

3. Boundary Value Analysis

It is generally observed that greater no of errors occur at the boundaries of the input domain rather than at the center.
This technique leads to the development of the test cases that exercise the boundary values.
Boundary value analysis is a test case design technique that complements equivalence partitioning.
Rather than selecting any element of an equivalence class, BVA leads to the selection of test cases at the "edges" of the class.
Rather than focusing solely on input conditions, BVA derives test cases from the output domain as well.

Guidelines

a. If the input condition specifies a range bounded by values a & b, then test cases must be designed with inputs as a & b, just above and below a & b.
b. Similarly if the input condition specifies a no of values, then test cases must be developed that exercise the minimum, maximum values & also the values just above and below the maximum and the minimum.
c. Apply the above guidelines also to the output i.e. design test cases such that you get the minimum and the maximum values of the outputs.
d. If the internal data structures have prescribed boundaries, then test the data structures at their boundaries (array limit of 100).

4. Comparison Testing

There are some situations (e.g., aircraft avionics, automobile braking systems) in which the reliability of software is absolutely critical.
In such applications redundant hardware and software are often used to minimize the possibility of error.
When redundant software is developed, separate software engineering teams develop independent versions of an application using the same specification.
In such situations, each version can be tested with the same test data to ensure that all provide identical output.

Then all versions are executed in parallel with real-time comparison of results to ensure consistency.
Comparison testing is not foolproof.
If the specification from which all versions have been developed is in error, all versions will likely reflect the error.
In addition, if each of the independent versions produces identical but incorrect results, condition testing will fail to detect the error.

5. Orthogonal Array Testing

Orthogonal array testing can be applied to problems in which the input domain is relatively small but too large to accommodate exhaustive testing.
The orthogonal array testing method is particularly useful in finding errors associated with region faults—an error category associated with faulty logic within a software component.

- It is a Black-box technique that enables the design of a reasonably small set of test cases that provide maximum test coverage.
- Focus is on categories of faulty logic likely to be present in the software component (without examining the code)
- Priorities for assessing tests using an orthogonal array

1. Detect and isolate all single mode faults
2. Detect all double mode faults
3. Multimode faults