Software Testing Techniques

Peter Lo

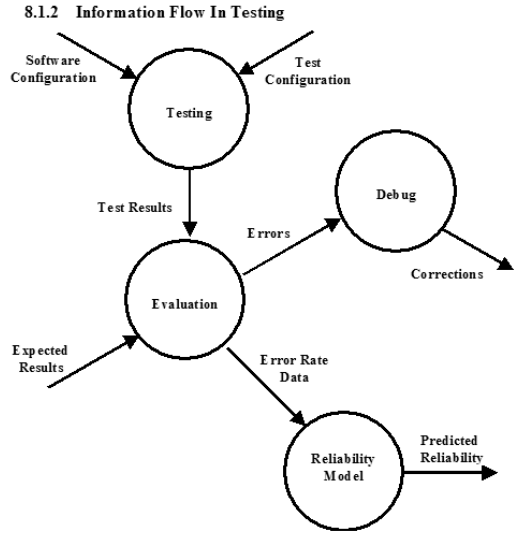
Software Testing Fundamentals

- Software Testing is a critical element of software quality assurance and represents the ultimate review of specification, design and coding.
- It concerned with the actively identifying errors in software

Testing Objectives

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

Information Flow in Testing



Information Flow in Testing

- Two classes of input are provided to the test process:
  - **A Software Configuration**: includes a Software Requirements Specification, a Design Specification, and source code.
  - **A Test Configuration:** includes a Test Plan and Procedure, any testing tools that are to be used, and test cases and their expected results.

Attributes of a "Good Test"

- A good test has a high probability of finding an error.
  - The tester must understand the software and attempt to develop a mental picture of how the software might fail.
- A good test is not redundant.
  - Testing time and resources are limited.
  - There is no point in conducting a test that has the same purpose as another test.
- A good test should be neither too simple nor too complex.
  - Side effect of attempting to combine a series of tests into one test case is that this approach may mask errors.

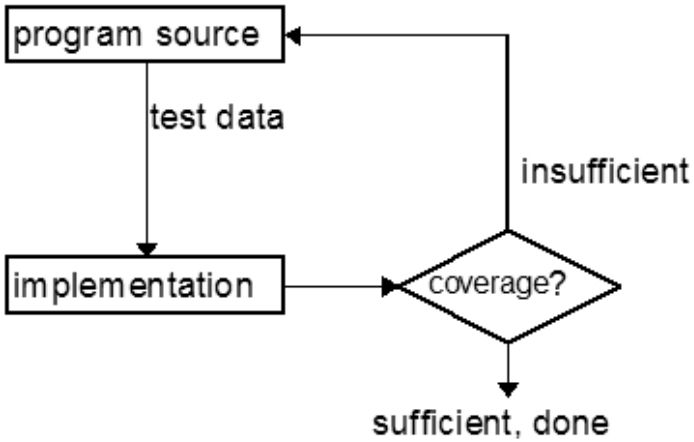
White Box Testing

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

Benefit of White Box Testing

- Using white box testing methods, the software engineer can derive test cases that:
  - Guarantee that all independent paths within a module have been exercised at least once;
  - Exercise all logical decisions on their true or false sides;
  - Execute all loops at their boundaries and within their operational bounds ; and
  - Exercise internal data structures to ensure their validity.

Process of White Box Testing



Process of White Box Testing

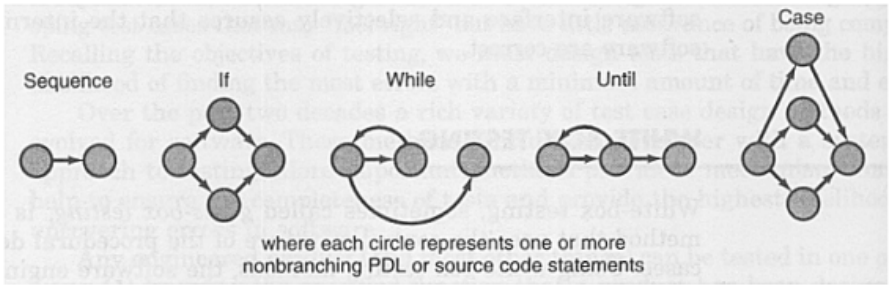
- Tests are derived from an examination of the source code for the modules of the program.
- These are fed as input to the implementation, and the execution traces are used to determine if there is sufficient coverage of the program source code

Reasons to Test the Program Logic

- Logic errors and incorrect assumptions are inversely proportional to the probability that a program path will evaluated
- Misconceptions that a logical path is unlikely to be executed when, in fact, it may be executed on a regular basis.
- Typographical errors are random.

Basis Path Testing

- This testing technique makes use of simple flow graph notation.



Basis Path Testing

- Basis Path Testing is a white box testing techniques proposed by Tom McCabe.
- Enables that test case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths.
- Test cases derived to exercise the basis set are guaranteed to execute every statement in the program at least one time during testing.

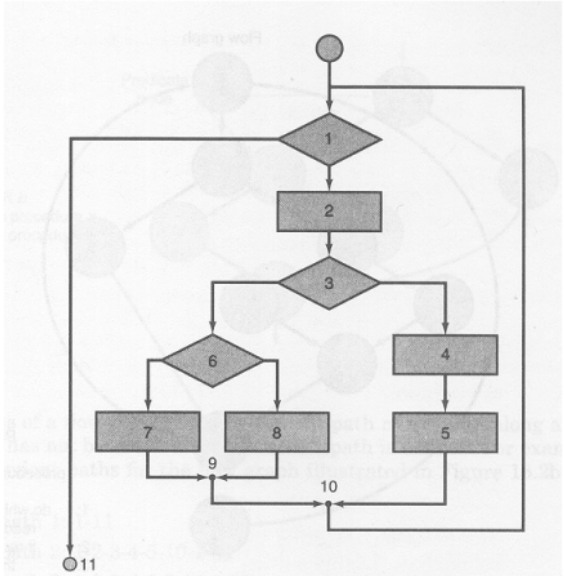
Basis Path Testing

- The basis path testing can be applied as a series of steps:
  - Using the design or code as foundation, draw a corresponding flow graph.
  - Determine the cyclomatic complexity of the resultant flow graph.
  - Determine a basis set of linearly independent paths.
  - Prepare test cases that will force execution of each path in the basis set.

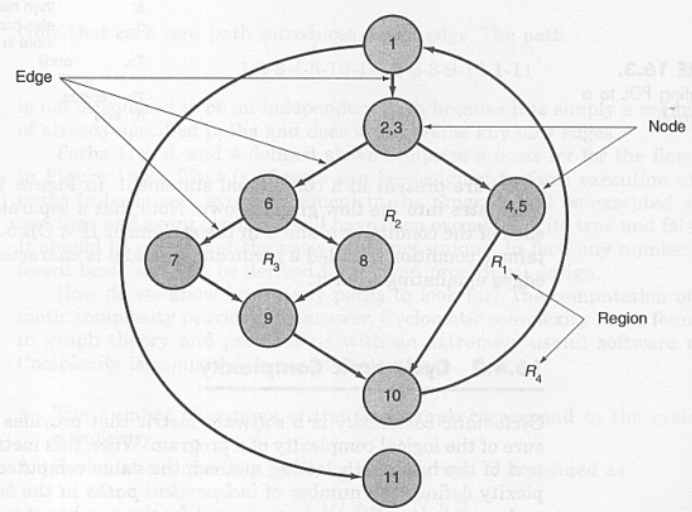
Cyclomatic Complexity V(G)

- Cyclomatic Complexity V(G) is a software metric that provides a quantitative measure of the logical complexity of a program.
- Complexity can be computed in one of several ways:
  - V(G) = Number of regions of the flow graph
  - V(G) = E – N + 2; where E is the number of flow graph edges and N is the number of flow graph nodes.
  - V(G) = P + 1 where P is the number of predicate nodes contained in the flow graph G.
- Predicate notes are characterized by two or more edges emanating from it.

Example



Answer: Graph



Answer: V(G) Calculation

- The flow has 4 regions (marked as R1, R2, R3 and R4); hence V(G) = 4
- V(G) = E - N + 2 = 11 edges - 9 nodes + 2 = 4
- V(G) = P + 1 = 3 Predicate Notes + 1 = 4 (Predicate nodes are: node (2,3), node (1), and node (6).

Answer: Independent Path

- Thus, the value of V(G) provides us with an upper bound for the number of independent paths ....(etc...)
- The set of independent paths are:
  - 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

Condition Testing

- Simple condition is a Boolean variable or relational expression
- Condition testing is a test case design method that exercises the logical conditions contained in a program module, and therefore focuses on testing each condition in the program.

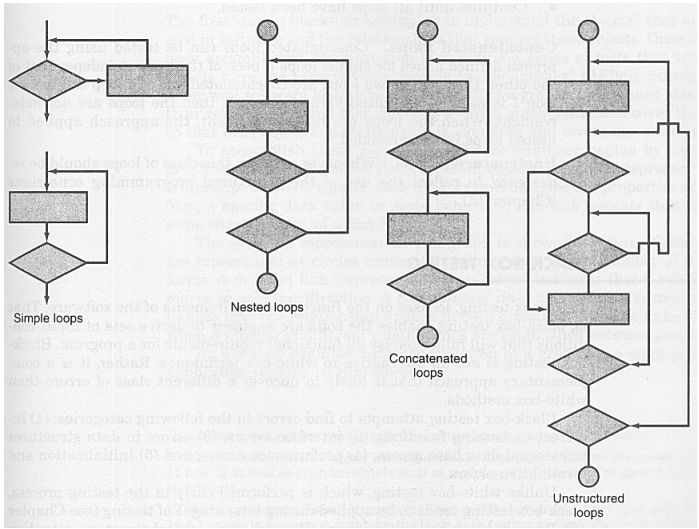
Data Flow Testing

- The Data flow testing method selects test paths of a program according to the locations of definitions and uses of variables in the program

Loop Testing

- Loop testing is a white-box testing technique that focuses exclusively on the validity of loop constructs
- Four classes of loops:
  1. Simple loops
  2. Concatenated loops
  3. Nested loops
  4. Unstructured loops

Loop Testing



Test Cases for Simple Loops

- Where n is the maximum number of allowable passes through the loop:
  - Skip the loop entirely
  - Only one pass through the loop
  - Two passes through the loop
  - m passes through the loop where m < n
  - n-1, n, n+1 passes through the loop

Test Cases for Nested Loops

1) Start at the innermost loops. 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 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
4) Continue until all loops have been tested

Test Cases for Concatenated Loops

- If each of the loops is independent of the others, perform simple loop tests for each loop
- If the loops are dependent, apply the nested loop tests

Test Cases for Unstructured Loops

- Whenever possible, redesign this class of loops to reflect the structured programming constructs

Black Box Testing

- Black box testing methods focus on the functional requirements of the software,
  - i.e. derives sets of input conditions that will fully exercise all functional requirements for a program.
- Black box or functional testing is based upon the specification of a module rather than the implementation of the module.
- Determines whether the required functionality is present, but since it is independent of the actual program code it cannot test parts of the program that are not covered by the specification.

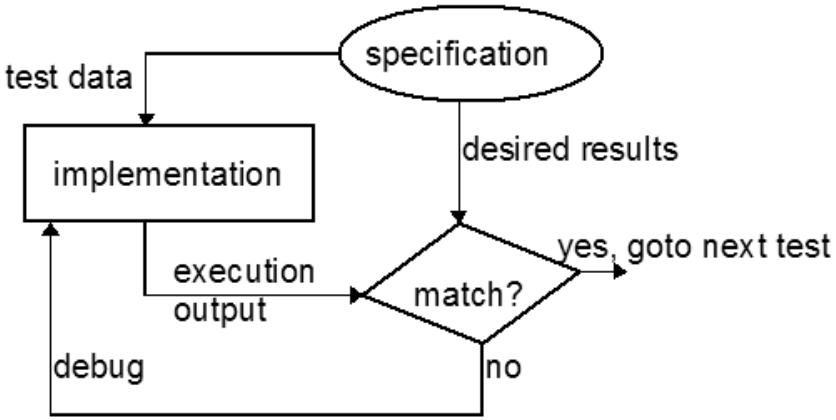
Black Box Testing

- Black box testing attempts to find errors in the following categories:
  - Incorrect or missing functions
  - Interface errors
  - Errors in data structures or external databases access
  - Performance errors
  - Initialization and termination errors.

Black Box Test Cases

- Test cases need to satisfy the following criteria:
  - Test cases that reduce, by a count that is greater than 1, the number of additional test cases that must be designed to achieve reasonable testing; and
  - Tests cases that tell us something about the presence or absence of classes of errors, rather than an error associated only with the specific test at hand.

Black Box Testing



Equivalence Partitioning

- Divides the input domain of a program into classes of data.
- Strives to define a test case that uncovers classes of errors
- Equivalence classes may be defined according to the following guidelines:
  - If an input condition specifies a range, one valid and two invalid equivalence classes are defined.
  - If an input condition requires a specific value, one valid and two invalid equivalence classes are defined.
  - If an input condition specifies a member of a set, one valid and one invalid class are defined
  - If an input condition is Boolean, one valid and one invalid class are defined

Equivalence Partitioning

- Input Data variable is Employment Age, as used in a company database system. Employment age is defined as an integer variable, and has an acceptable range of 16 to 65 (years old).
- Since this is a range, there will be one valid, and two invalid equivalence classes.
- **The Valid Class:** Employment age = from 16 to 65, inclusive. An example of a test case falling into this class would be Employment age = 30.
- **The First Invalid Class:** Employment age < 16. An example of a test case falling into this class would be Employment age = 10.
- **The Second Invalid Class:** Employment age > 65. An example of a test case falling into this class would be Employment age = 80.

Boundary Value Analysis

- Boundary value analysis (BVA) select test cases at the "edges", and thus exercises bounding values.
- BVA leads to the selection of test cases at the “edges” of the class.

Boundary Value Analysis

- Guidelines of designing test cases
  - If an input condition specifies a range bounded by values a and b, test cases should be designed with values a and b, just above and below a and b, respectively
  - If an input condition specifies a number of values, test cases should be developed that exercise the minimum and maximum numbers. Values just above and below minimum and maximum are also tested.

Boundary Value Analysis

- Input Data variable is Employment Age, as used in a company database system. Employment age is defined as an integer variable, and has an acceptable range of 16 to 65 (years old).
- The two “edges” are 16 and 65.
- Hence, the six test cases would be:
- Employment Age = 15, 16, 17, and 64, 65, and 66.

Comparison Testing

- Used when the reliability of software is absolutely critical.
- Multiple and independent versions of software is developed for critical applications, even when only a single version will be used in the delivered computer-based system
- Each version is tested with the same test data to ensure that all provide identical output.

Comparison Testing

- All the versions are executed in parallel with a real-time comparison of results to ensure consistency.
  - If the output from each version is the same, then it is assumed that all implementations are correct.
  - If the output is different, each of the applications is investigated to determine if a defect in one or more versions is responsible for the difference.

Automated Testing Tools

- Code auditors
- Assertion processors
- Test file generators
- Test data generators
- Test verifiers
- Output comparators

Code Auditors

- These special-purpose filters are used to check the quality of software to ensure that it meets minimum coding standards.

Assertion Processors

- These pre-processors / postprocessors systems are employed to tell whether programmer-supplied claims, called assertions, about a program's behavior are actually met during real program executions.

Test File Generators

- These processors generate, and fill with predetermined values, typical input files for programs that are undergoing testing.

Test Data Generators

- These automated analysis systems assist a user in selecting test data that make a program behave in a particular fashion.

Test Verifiers

- These tools measure internal test coverage, often expressed in terms that are related to the control structure of the test object, and report the coverage value to the quality assurance expert

Output Comparators

- This tool makes it possible to compare one set of outputs from a program with another (previously archived) set to determine the difference between them.

Basis Path Testing: An Exercise

```
1.    While (x>0) and (data <> 0) do
2.        if (x>5) then
3.                y = x + 3
4.        else
5.                y = 5
6.        endif
7.        x = x - 1
8.        data = data - 1
9.    end While
10.   If data = 0 then
11.       write ('Goodbye')
12.   else
13.       write ('Error')
14.   endif
15.   write ('End of Program')
```


Basis Path Testing: An Exercise

- Carefully consider the pseudo-code above, and draw the corresponding Flow Graph. In your answer, also indicate the pseudo-code statement that each node is representing.
- Calculate the value of Cyclomatic Complexity V(G). Calculate this number using the Number of Regions method.
- Using the Cyclomatic Complexity, design a set of test cases that will adequately test the entire program.