Software Engineering

Third Class

Lecture 13

SOFTWARE TESTING TECHNIQUES

13.1 Software Testing Fundamentals

Testing presents an interesting anomaly for the software engineer. During earlier software engineering activities, the engineer attempts to build software from an abstract concept to a tangible product. Now comes testing. The engineer creates a series of test cases that are intended to "demolish" the software that has been built. In fact, testing is the one step in the software process that could be viewed (psychologically, at least) as destructive rather than constructive.

13.1.1 Testing Objectives

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.

Our objective is to design tests that systematically uncover different classes of errors and to do so with a minimum amount of time and effort.

13.1.2 Testing Principles

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

- **All tests should be traceable to customer requirements.** 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.

- **The Pareto principle applies to software testing.** Stated simply, the Pareto principle implies that 80 percent of all errors uncovered during testing will likely be traceable to 20 percent of all program components. The problem, of course, is to isolate these suspect components and to thoroughly test them.
- **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.

And what about the tests themselves? Kaner, Falk, and Nguyen suggest the following attributes of a "good" test:

1. A good test has a high probability of finding an error. To achieve this goal, the tester must understand the software and attempt to develop a mental picture of how the software might fail
2. 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. Every test should have a different purpose (even if it is subtly different).
3. A good test should be "best of breed" . In a group of tests that have a similar intent, time and resource limitations may mitigate toward the execution of only a subset of these tests. In such cases, the test that has the highest likelihood of uncovering a whole class of errors should be used.

4. A good test should be neither too simple nor too complex. Although it is sometimes possible to combine a series of tests into one test case, the possible side effects associated with this approach may mask errors. In general, each test should be executed separately.

13.2 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. Yet, software engineers often treat testing as an afterthought, developing test cases that may "feel right" but have little assurance of being complete. Recalling the objectives of testing, we must design tests that have the highest likelihood of finding the most errors with a minimum amount of time and effort.

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 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.

13.3 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.

13.3.1 BASIS PATH TESTING

Basis path testing is a white-box testing technique first proposed by Tom McCabe. The basis path method enables the 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.

Flow Graph Notation

Before the basis path method can be introduced, a simple notation for the representation of control flow, called a *flow graph* (or *program graph*) must be introduced. The flow graph depicts logical control flow using the notation illustrated in Figure 1. Each structured construct (Lecture 11) has a corresponding flow graph symbol.

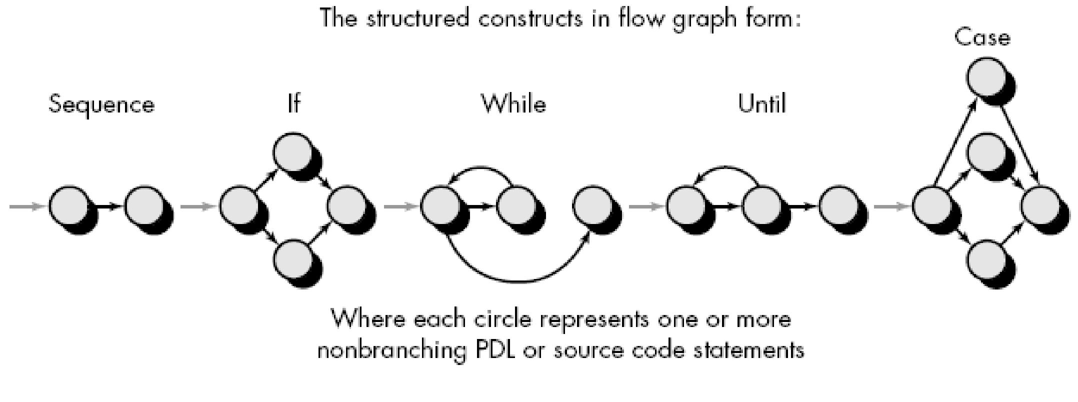
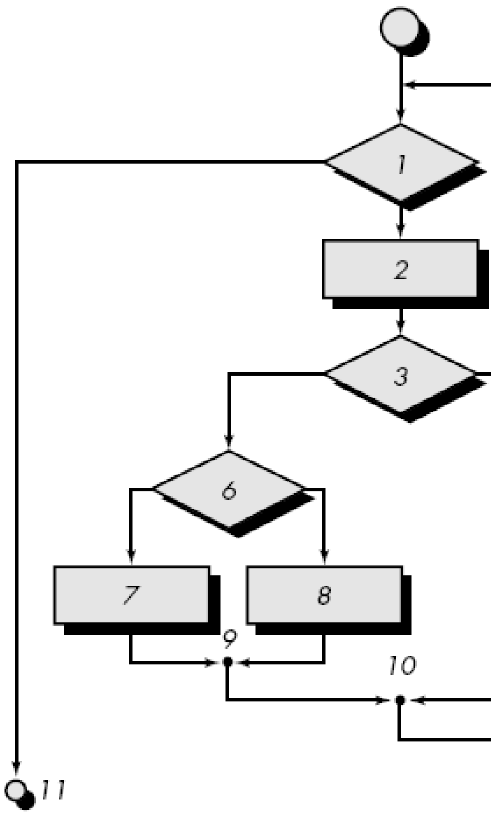
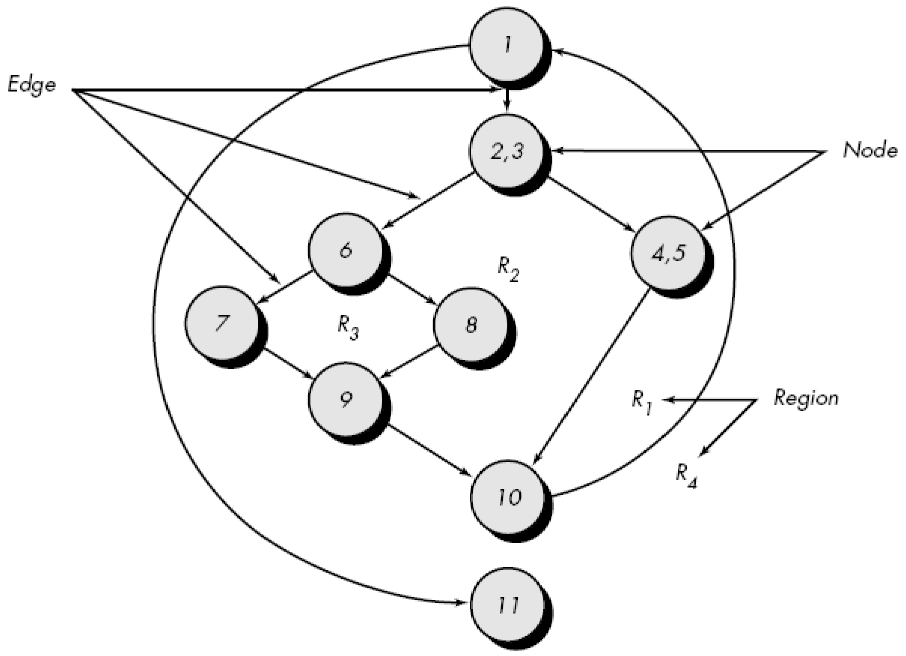

The structured constructs in flow graph form:
Sequence
If
While
Until
Case
Where each circle represents one or more nonbranching PDL or source code statements

Figure 1: Flow graph notation

To illustrate the use of a flow graph, we consider the procedural design representation in Figure 2A.



(A)



(B)

Here, a flowchart is used to depict program control structure.
Figure 2B maps the flowchart into a corresponding flow graph (assuming that no compound conditions are contained in the decision diamonds of the flowchart).
Referring to Figure 2 B, each circle, called a *flow graph node,* represents one or more procedural statements. A sequence of process boxes and a decision diamond can map into a single node. The arrows on the flow graph, called *edges* or *links,* represent flow of control and are analogous to flowchart arrows. An edge must terminate at a node, even if the node does not represent any procedural statements (e.g., see the symbol for the if-then-else construct). Areas bounded by edges and nodes are called *regions.* When counting regions, we include the area outside the graph as a region.
When compound conditions are encountered in a procedural design, the generation of a flow graph becomes slightly more complicated. A compound condition occurs when one or more Boolean operators (logical OR, AND, NAND, NOR) is present in a conditional statement.

Cyclomatic Complexity

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. When used in the context of the basis path testing method, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program and provides us with an upper bound for the number of tests that must be conducted to ensure that all statements have been executed at least once.
An *independent path* is any path through the program that introduces at least one new set of processing statements or a new condition. When stated in terms of a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined. For example, a set of independent paths for the flow graph illustrated in Figure 2B is

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.

Paths 1, 2, 3, and 4 constitute a *basis set* for the flow graph in Figure 2B. That is, if tests can be designed to force execution of these paths (a basis set), every statement in the program will have been guaranteed to be executed at least one time and every condition will have been executed on its true and false sides. It should be noted that the basis set is not unique. In fact, a number of different basis sets can be derived for a given procedural design.

How do we know how many paths to look for? The computation of cyclomatic complexity provides the answer.

Cyclomatic complexity has a foundation in graph theory and provides us with an extremely useful software metric. Complexity is computed in one of three ways:

1. The number of regions of the flow graph correspond to the cyclomatic complexity.

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.

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.

Referring once more to the flow graph in Figure 2B, the cyclomatic complexity can be computed using each of the algorithms just noted:

1. The flow graph has four regions.

2. $V(G)$ = 11 edges _ 9 nodes + 2 = 4.

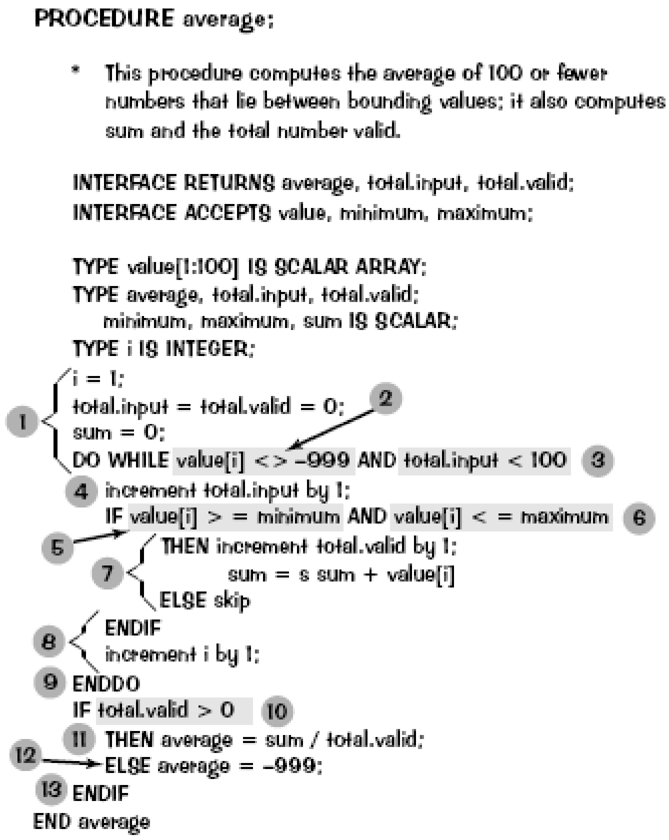
3. $V(G)$ = 3 predicate nodes + 1 = 4.

Therefore, the cyclomatic complexity of the flow graph in Figure 2B is 4.

More important, the value for $V(G)$ provides us with an upper bound for the number of independent paths that form the basis set and, by implication, an upper bound on the number of tests that must be designed and executed to guarantee coverage of all program statements.

Deriving Test Cases

The basis path testing method can be applied to a procedural design or to source code. In this section, we present basis path testing as a series of steps. The procedure *average*, depicted in PDL in Figure 3, will be used as an example to illustrate each step in the test case design method.



```
PROCEDURE average;

   *  This procedure computes the average of 100 or fewer
      numbers that lie between bounding values; it also computes the
      sum and the total number valid.

   INTERFACE RETURNS average, total.input, total.valid;
   INTERFACE ACCEPTS value, minimum, maximum;

   TYPE value[1:100] IS SCALAR ARRAY;
   TYPE average, total.input, total.valid;
      minimum, maximum, sum IS SCALAR;
   TYPE i IS INTEGER;
   i = 1;
   total.input = total.valid = 0;
   sum = 0;
   DO WHILE value[i] <> -999 AND total.input < 100
      increment total.input by 1;
      IF value[i] > = minimum AND value[i] < = maximum
         THEN increment total.valid by 1;
                 sum = s sum + value[i]
         ELSE skip
      ENDIF
      increment i by 1;
   ENDDO
   IF total.valid > 0
      THEN average = sum / total.valid;
      ELSE average = -999;
   ENDIF
END average
```

Figure 3:PDL for test case design with nodes identified

Note that *average*, although an extremely simple algorithm, contains compound conditions and loops. The following steps can be applied to derive the basis set:

1. Using the design or code as a foundation, draw a corresponding flow graph.

A flow graph is created using the symbols and construction rules presented earlier in the lecture . Referring to the PDL for average in Figure 3, a flow graph is created by numbering those PDL statements that will be mapped into corresponding flow graph nodes. The corresponding flow graph is in Figure 4.

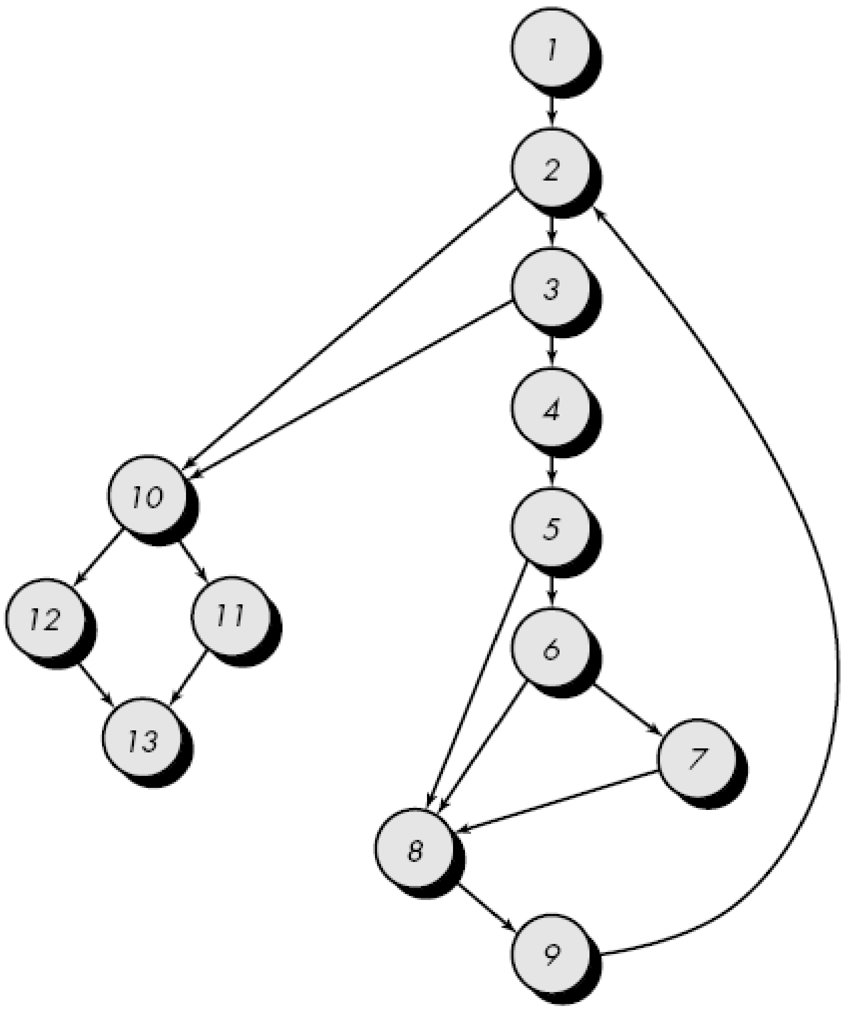


Figure 4: Flow graph for the procedure average

2. **Determine the cyclomatic complexity of the resultant flow graph.** The cyclomatic complexity, $V(G)$, is determined by applying the algorithms earlier. It should be noted that $V(G)$ can be determined without developing a flow graph by counting all conditional statements in the PDL (for the procedure *average,* compound conditions count as two) and adding 1. Referring to Figure 4,

$V(G)$ = 6 regions

$V(G)$ = 17 edges - 13 nodes + 2 = 6

$V(G)$ = 5 predicate nodes + 1 = 6

3. Determine a basis set of linearly independent paths. The value of $V(G)$ provides the number of linearly independent paths through the program control structure. In the case of procedure *average,* we expect to specify six paths:

path 1: 1-2-10-11-13

path 2: 1-2-10-12-13

path 3: 1-2-3-10-11-13

path 4: 1-2-3-4-5-8-9-2-. . .

path 5: 1-2-3-4-5-6-8-9-2-. . .

path 6: 1-2-3-4-5-6-7-8-9-2-. . .

The ellipsis (. . .) following paths 4, 5, and 6 indicates that any path through the remainder of the control structure is acceptable. It is often worthwhile to identify predicate nodes as an aid in the derivation of test cases. In this case, nodes 2, 3, 5, 6, and 10 are predicate nodes.

4-Prepare test cases that will force execution of each path in the basis set. Data should be chosen so that conditions at the predicate nodes are appropriately set as each path is tested. Each test case is executed and compared to expected results. Once all test cases have been completed, the tester can be sure that all statements in the program have been executed at least once.

13.3.2 Loop Testing

Loops are the cornerstone for the vast majority of all algorithms implemented in software. And yet, we often pay them little heed while conducting software tests. *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 (Figure 5).

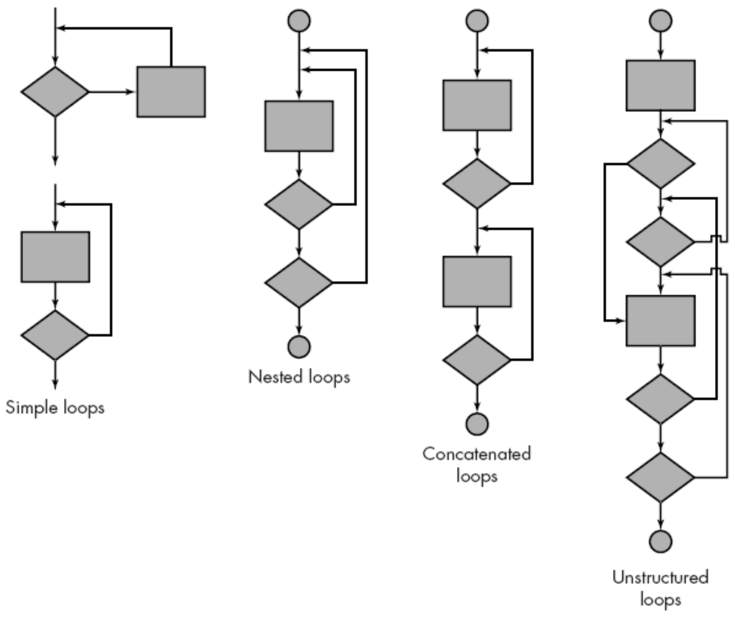


Figure 5: Classes of loops

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.

Nested loops. If we were to extend the test approach for simple loops to nested loops, the number of possible tests would grow geometrically as the level of nesting increases. This would result in an impractical number of tests. An approach that will help to reduce the number of tests:

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.

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.