Chapter 18

- **Testing Conventional Applications**

Slide Set to accompany

Software Engineering: A Practitioner's Approach, 7/e

by Roger S. Pressman

Testability

- Operability—it operates cleanly
- Observability—the results of each test case are readily observed
- Controllability—the degree to which testing can be automated and optimized
- Decomposability—testing can be targeted
- Simplicity—reduce complex architecture and logic to simplify tests
- Stability—few changes are requested during testing
- Understandability—of the design

What is a “Good” Test?

- A good test has a high probability of finding an error
- A good test is not redundant.
- A good test should be “best of breed”
- A good test should be neither too simple nor too complex

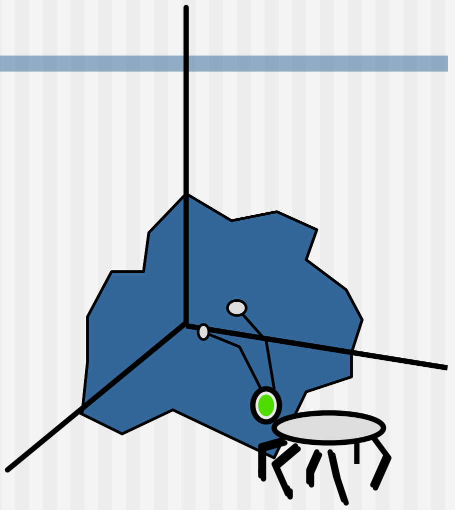
Internal and External Views

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

Test Case Design

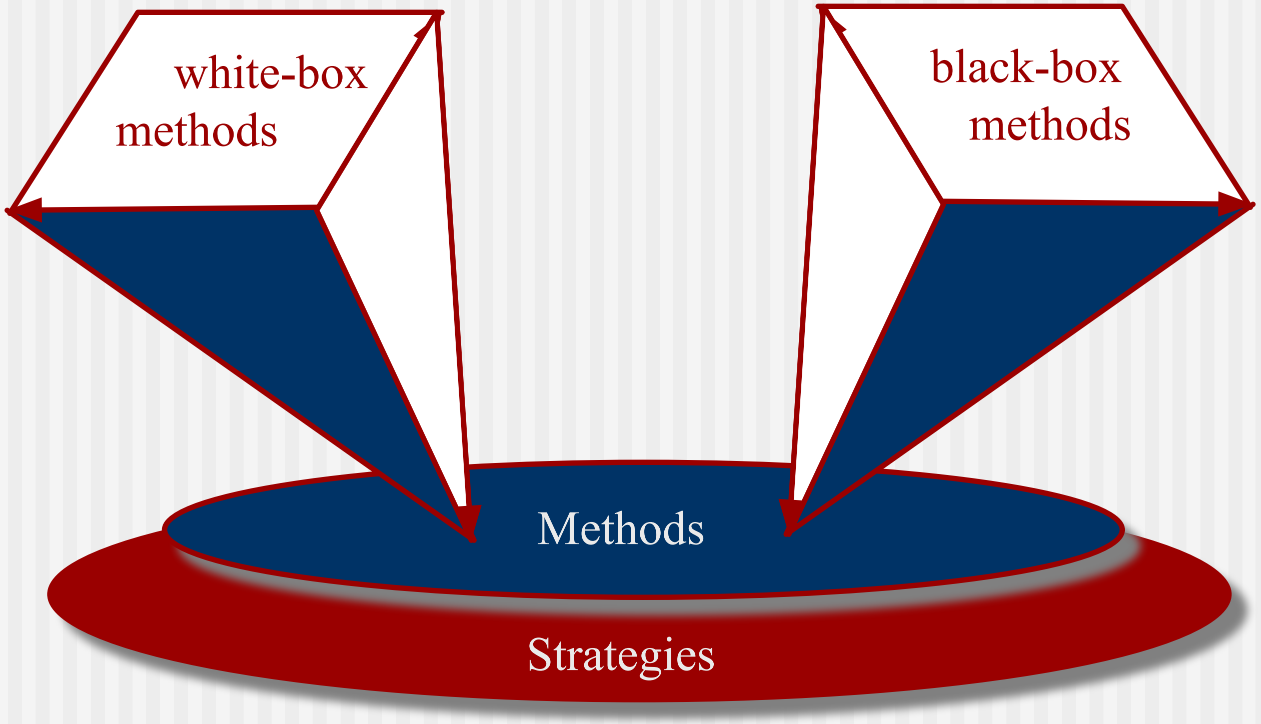
"Bugs lurk in corners and congregate at boundaries ..."

Boris Beizer

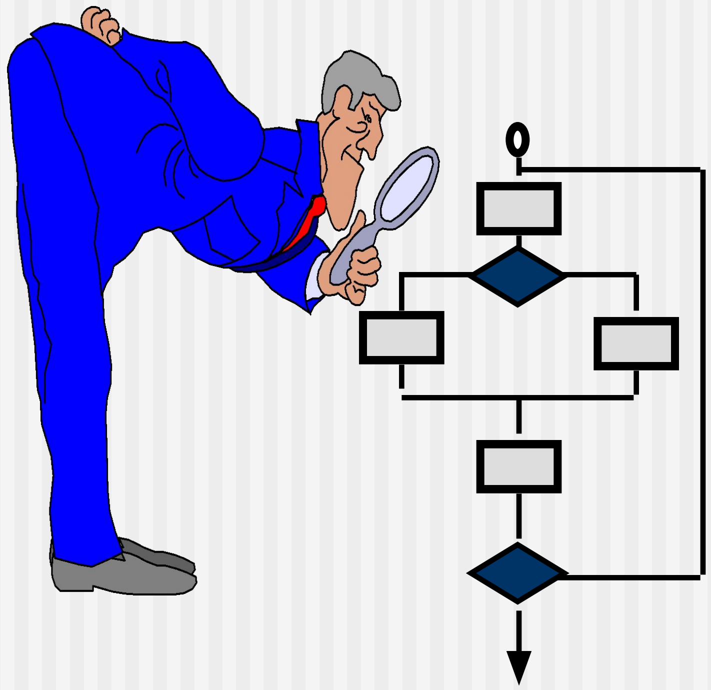


OBJECTIVE	**to uncover errors**
CRITERIA	**in a complete manner**
CONSTRAINT	**with a minimum of effort and time**

Software Testing

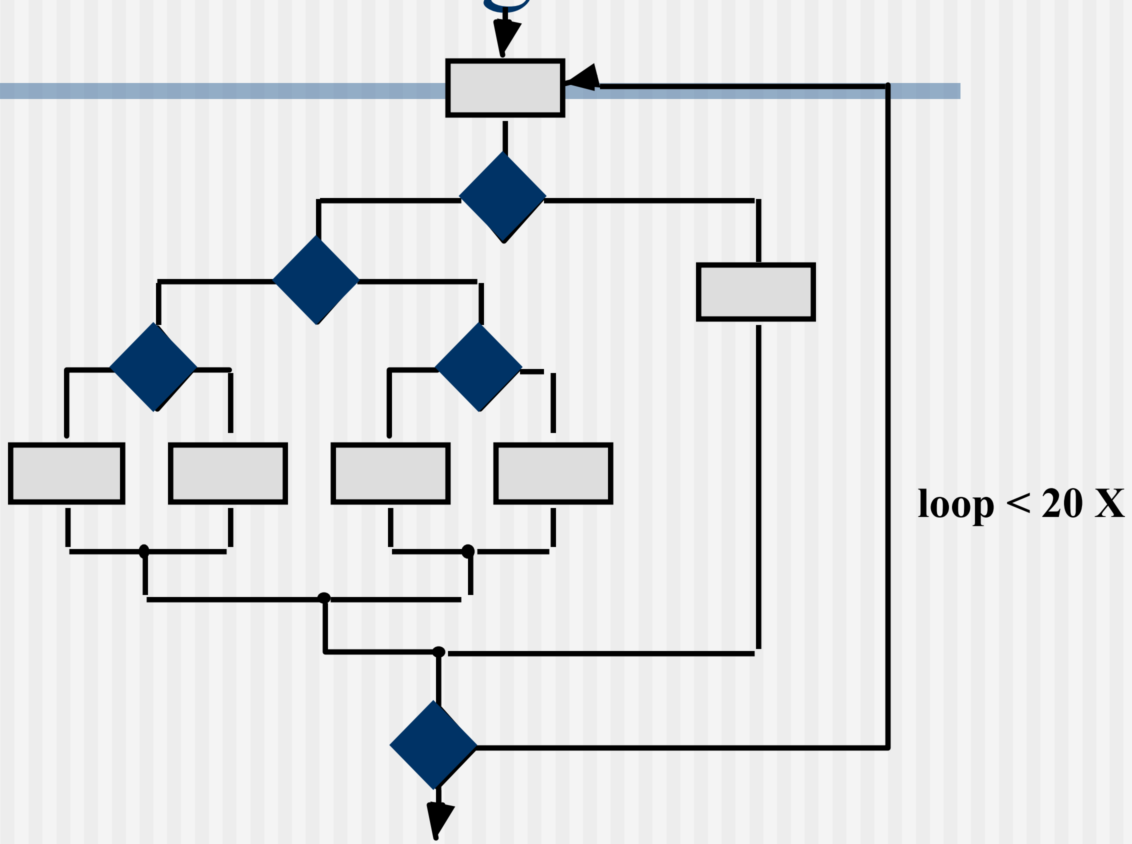


White-Box Testing



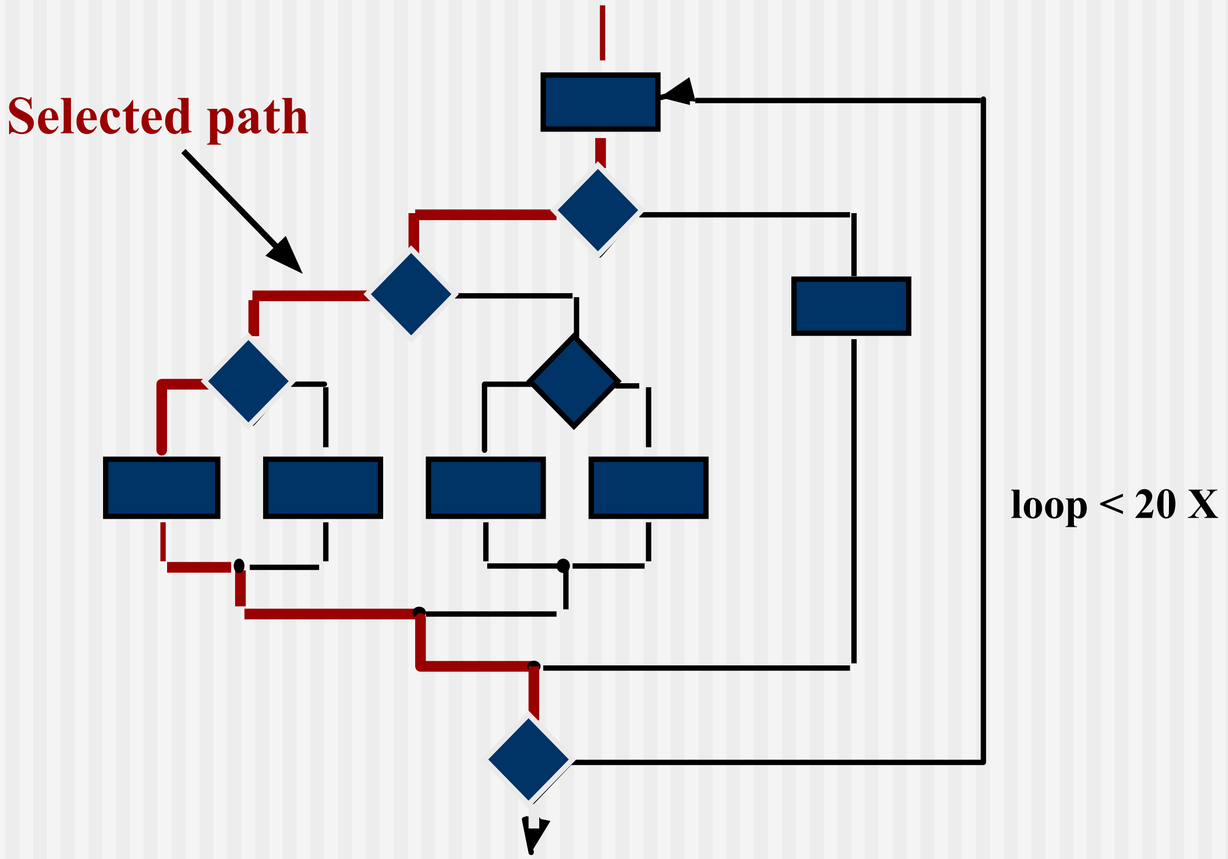
... our goal is to ensure that all statements and conditions have been executed at least once ...

Exhaustive Testing

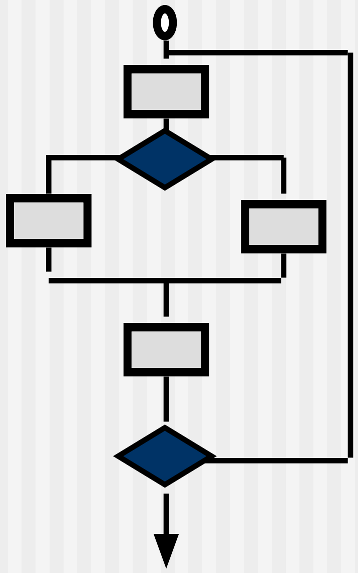


There are 10^{14} possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!

Selective Testing



White-Box Testing

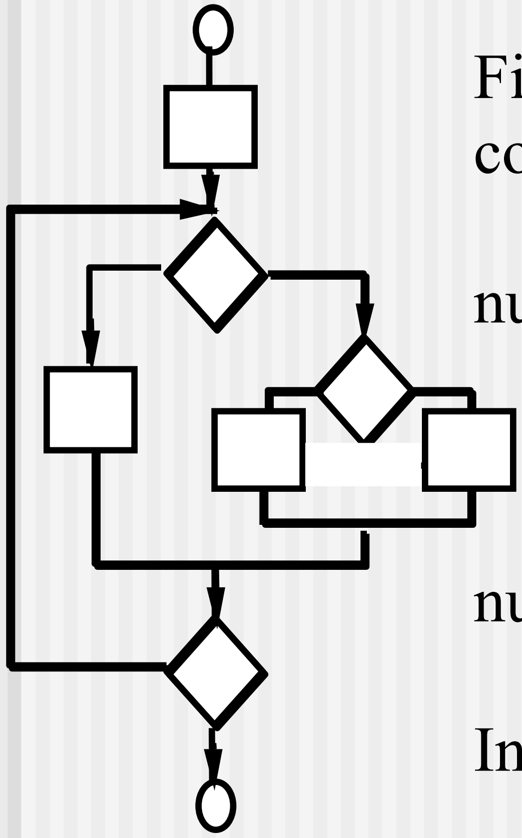


... our goal is to ensure that all statements and conditions have been executed at least once ...

Why Cover?

- logic errors and incorrect assumptions are inversely proportional to a path's execution probability
- we often <u>believe</u> that a path is not likely to be executed; in fact, reality is often counter intuitive
- typographical errors are random; it's likely that untested paths will contain some

Basis Path Testing



First, we compute the cyclomatic complexity:

number of simple decisions + 1

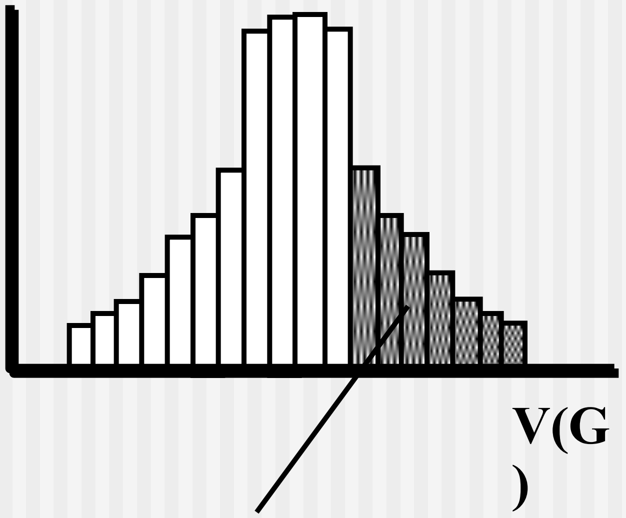
or

number of enclosed areas + 1

In this case, V(G) = 4

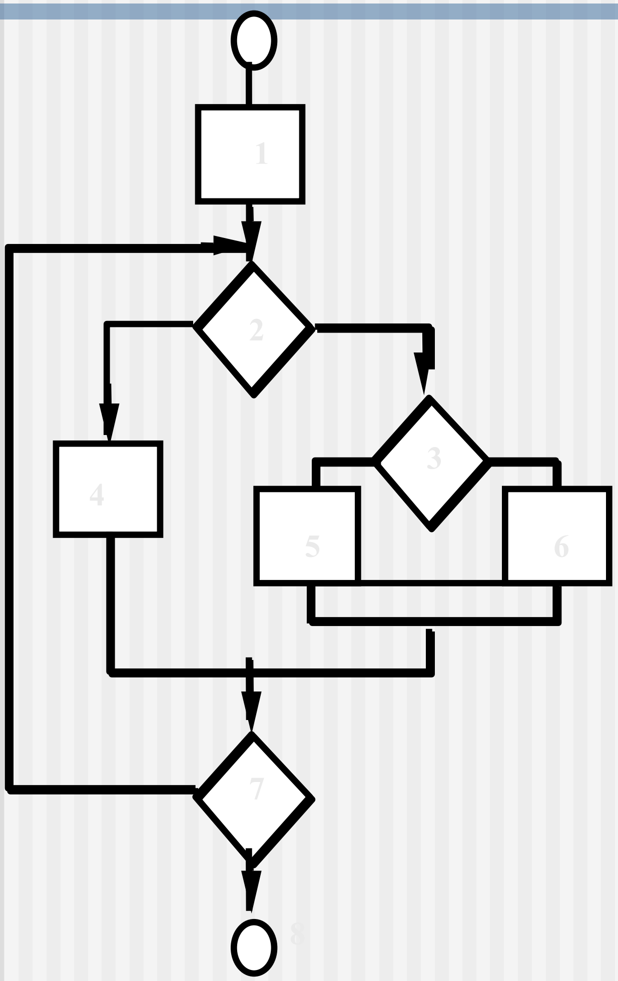
Cyclomatic Complexity

A number of industry studies have indicated that the higher V(G), the higher the probability or errors.



modules in this range are more error prone

Basis Path Testing



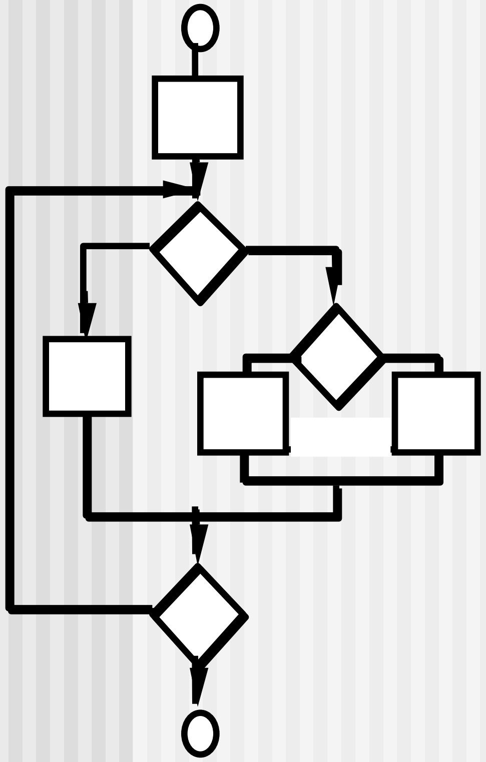
Next, we derive the independent paths:

Since V(G) = 4,
there are four paths

Path 1:	**1,2,3,6,7,8**
Path 2:	**1,2,3,5,7,8**
Path 3:	**1,2,4,7,8**
Path 4:	**1,2,4,7,2,4,...7,8**

Finally, we derive test cases to exercise these paths.

Basis Path Testing Notes

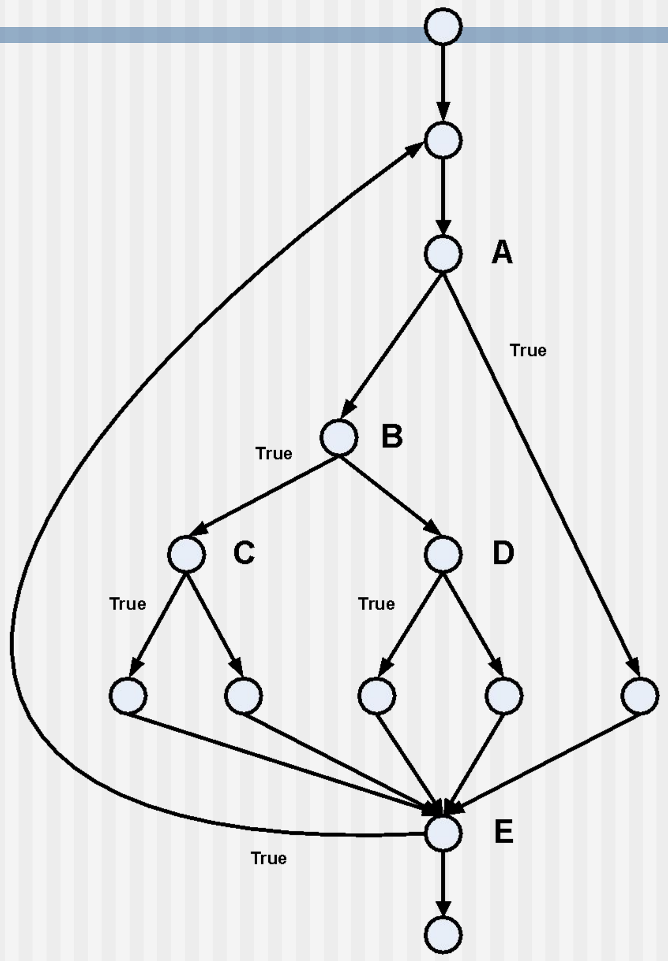


- **you don't need a flow chart, but the picture will help when you trace program paths**
- **count each simple logical test, compound tests count as 2 or more**
- **basis path testing should be applied to critical modules**

Example Code Fragment

```
Do
{
   if (A) then {...};
   else {
      if (B) then {
                  if (C) then {...};
                  else {…}
      }
      else if (D) then {...};
             else {...};
   }
}
While (E);
```

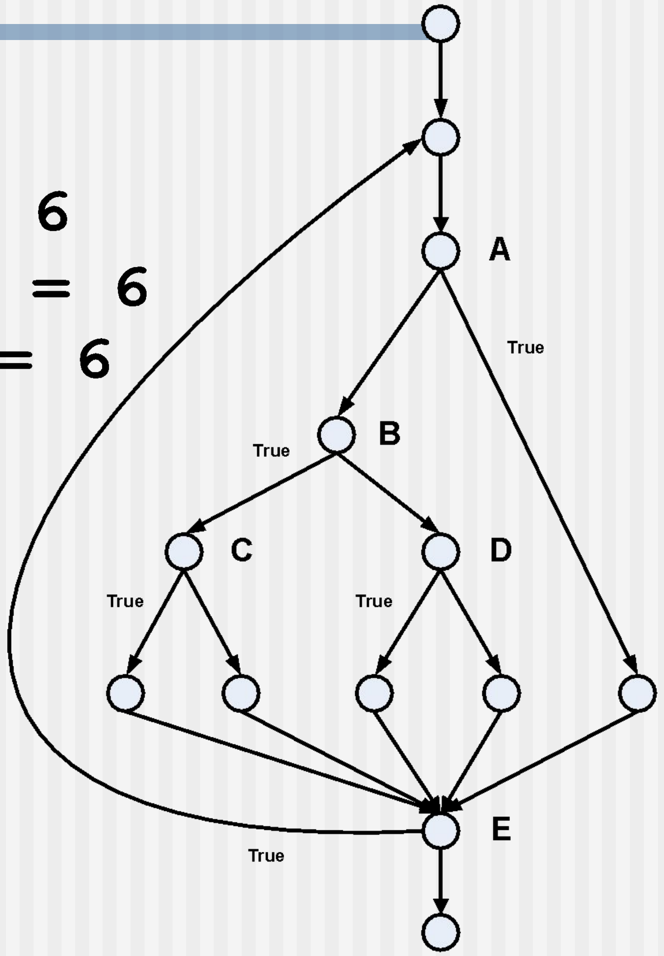
Example Control Flow Graph



Source: The Art of Software Testing – Glenford Myers

Calculating Complexity

(a) # regions + 1 = 5+1 = 6
(b) # decisions + 1 = 5+1 = 6
(c) L – N + 2P = 17-13+2 = 6



Deriving Test Cases

- *Summarizing:*
  - Using the design or code as a 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.

Graph Matrices

- A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on a flow graph
- Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.
- By adding a *link weight* to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing

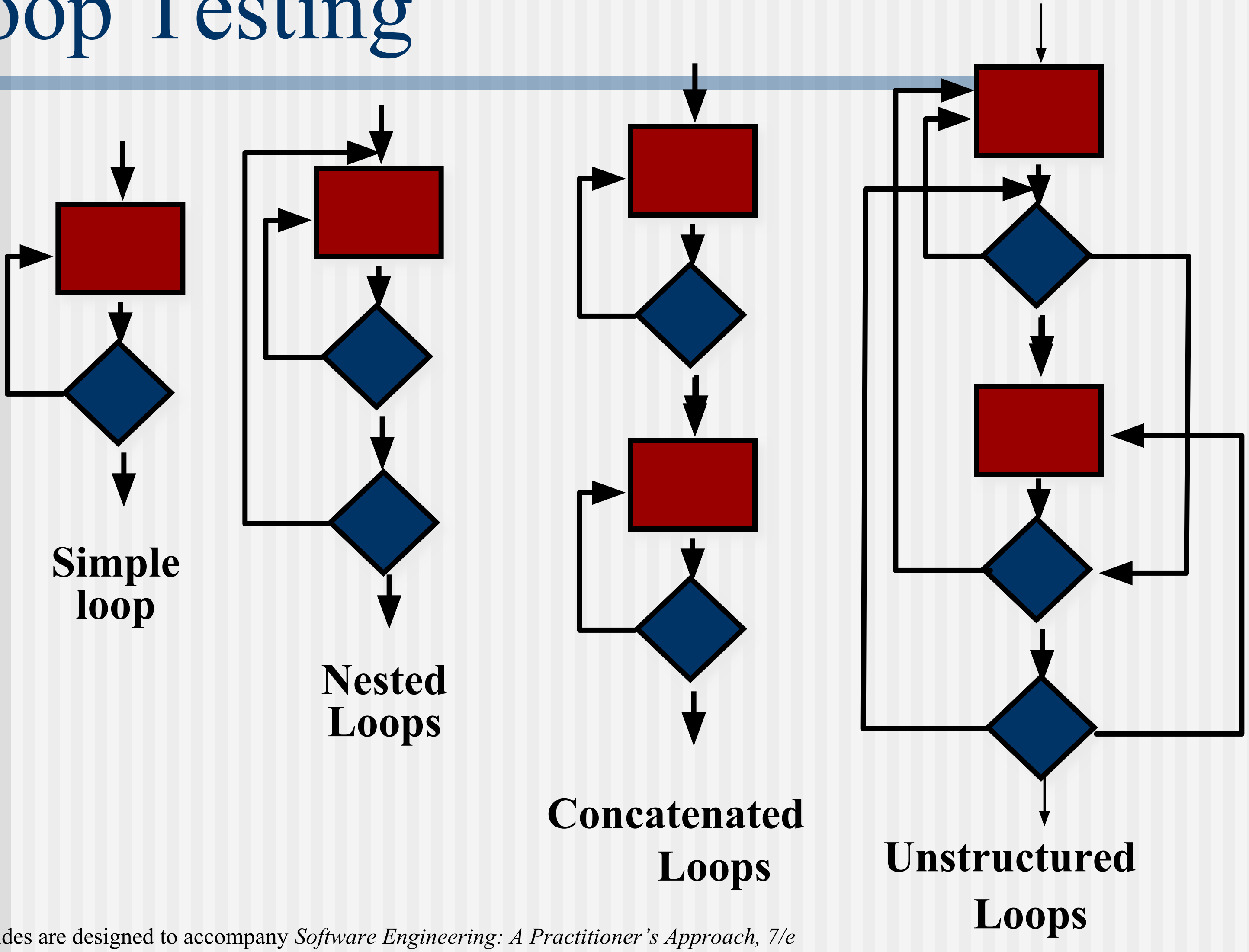
Control Structure Testing

- Condition testing — a test case design method that exercises the logical conditions contained in a program module
- Data flow testing — selects test paths of a program according to the locations of definitions and uses of variables in the program

Data Flow Testing

- The data flow testing method [Fra93] selects test paths of a program according to the locations of definitions and uses of variables in the program.
  - Assume that each statement in a program is assigned a unique statement number and that each function does not modify its parameters or global variables. For a statement with *S* as its statement number
    - DEF(*S*) = {*X* | statement *S* contains a definition of *X*}
    - USE(*S*) = {*X* | statement *S* contains a use of *X*}
  - A *definition-use (DU) chain* of variable X is of the form [*X, S, S′*], where *S* and *S′* are statement numbers, *X* is in DEF(*S*) and USE(*S′*), and the definition of *X* in statement *S* is live at statement *S′*

Loop Testing



Loop Testing: Simple Loops

Minimum conditions—Simple Loops

1. **skip the loop entirely**
2. **only one pass through the loop**
3. **two passes through the loop**
4. **m passes through the loop m < n**
5. **(n-1), n, and (n+1) passes through the loop**

where n is the maximum number of allowable passes

Loop Testing: Nested Loops

Nested Loops

Start at the innermost loop. Set all outer loops to their minimum iteration parameter values.

Test the min+1, typical, max-1 and max for the innermost loop, while holding the outer loops at their minimum values.

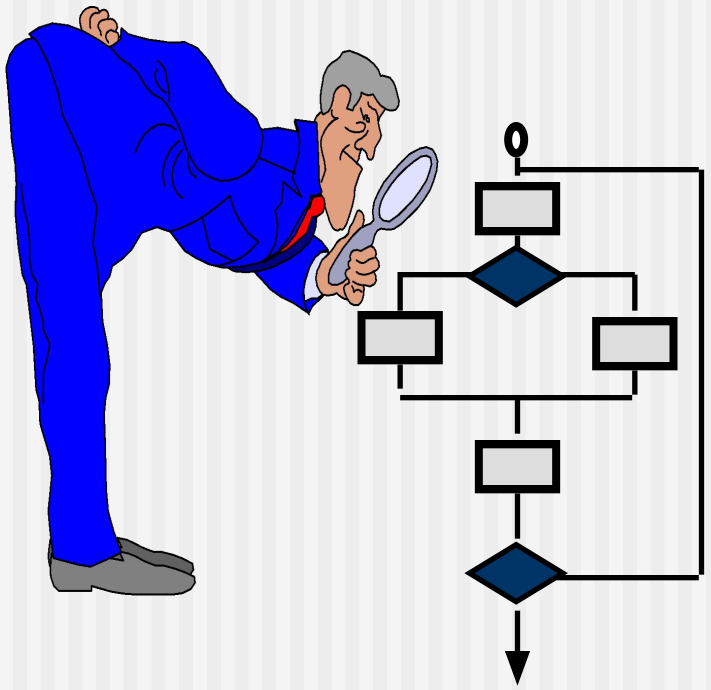
Move out one loop and set it up as in step 2, holding all other loops at typical values. Continue this step until the outermost loop has been tested.

Concatenated Loops

```
If the loops are independent of one another
   then treat each as a simple loop
   else* treat as nested loops
endif*
```

for example, the final loop counter value of loop 1 is used to initialize loop 2.

White-Box Testing



... our goal is to ensure that all statements and conditions have been executed at least once ...

White Box testing

1. Basic Concepts:

- Fault based testing
- Coverage based testing
- Testing criterion for coverage-based testing
- Stronger vs. weaker testing

Fault-based testing:

- Design test cases that focus on discovering certain types of faults.
- Example: Mutation testing.

Coverage-based testing:

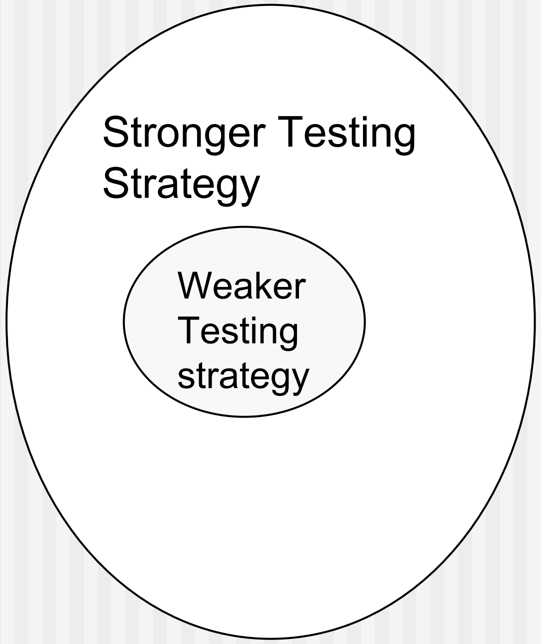
- Design test cases so that certain program elements are executed (or covered).
- Example: statement coverage, path coverage, etc.

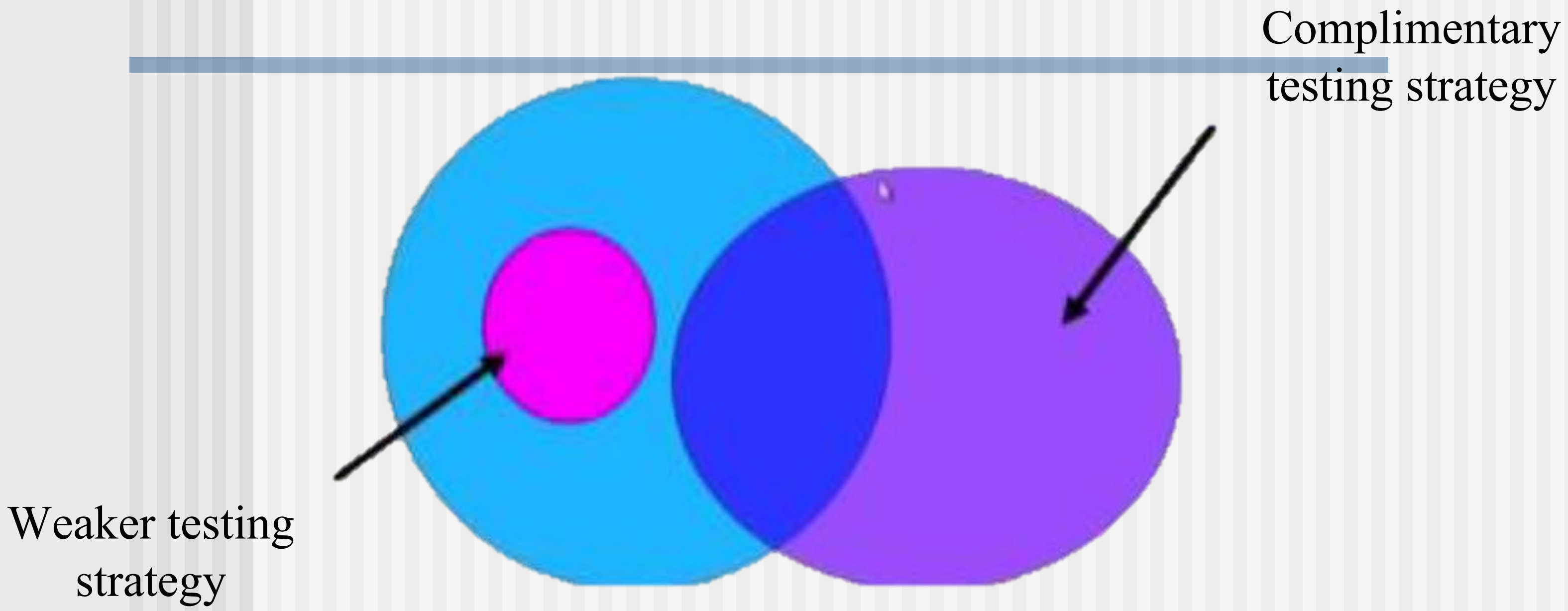
- Testing criterion for coverage-based testing

The set of specific program elements that a testing strategy targets to execute is called the testing criterion of the strategy.

- Stronger vs. weaker testing

A white box testing strategy is said to be stronger than another strategy, if the stronger testing strategy covers all program elements covered by the weaker testing strategy and the stronger strategy additionally covers at least one program element that is not covered by the weaker strategy.


Stronger Testing Strategy
Weaker Testing strategy


Complimentary testing strategy
Weaker testing strategy

2. Statement Coverage

- Statement coverage methodology:
  - Design test cases so that every statement in the program is executed at least once.

Statement Coverage

- The principal idea:
  - Unless a statement is executed,
  - We have no way of knowing if an error exists in that statement.

Statement Coverage Criterion

- Observing that a statement behaves properly for one input value:
  - No guarantee that it will behave correctly for all input values.

Example

```
int f1(int x, int y){
1 while (x != y){
2    if (x>y) then
3         x=x-y;
4    else y=y-x;
5 }
6 return x;        }
```

Euclid's GCD Algorithm

Euclid's GCD Computation Algorithm

- By choosing the test set {(x=3,y=3),(x=4,y=3),(x=3,y=4)}
  - All statements are executed at least once.

2. Branch Coverage

- Test cases are designed such that:
  - Different branch conditions
    - Given true and false values in turn.

Branch Coverage

- Branch testing guarantees statement coverage:
  - A stronger testing compared to the statement coverage-based testing.

Example

```
int f1(int x,int y){
1 while (x != y){
2     if (x>y) then
3           x=x-y;
4     else y=y-x;
5 }
6 return x;        }
```

Example

- Test cases for branch coverage can be:
- {(x=3,y=3),(x=3,y=2), (x=4,y=3), (x=3,y=4)}

3. Condition Coverage

- Test cases are designed such that:
  - Each component of a composite conditional expression
    - Given both true and false values.

Example

- Consider the conditional expression
  - ((c1.and.c2).or.c3):
- Each of c1, c2, and c3 are exercised at least once,
  - i.e. given true and false values.

Condition coverage

- Consider a boolean expression having n components:
  - For condition coverage we require 2^n test cases.
- Condition coverage-based testing technique:
  - Practical only if n (the number of component conditions) is small.