Chapter 23

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

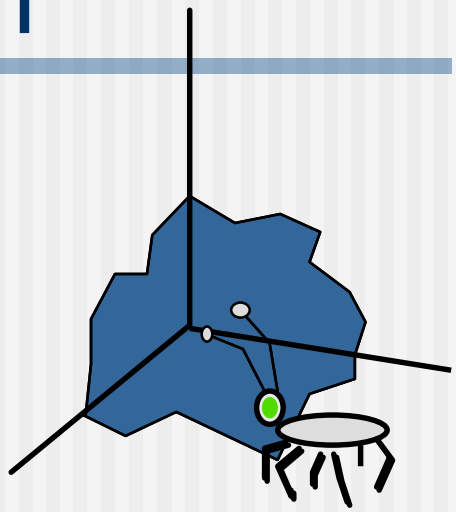
Internal and External Views

- 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

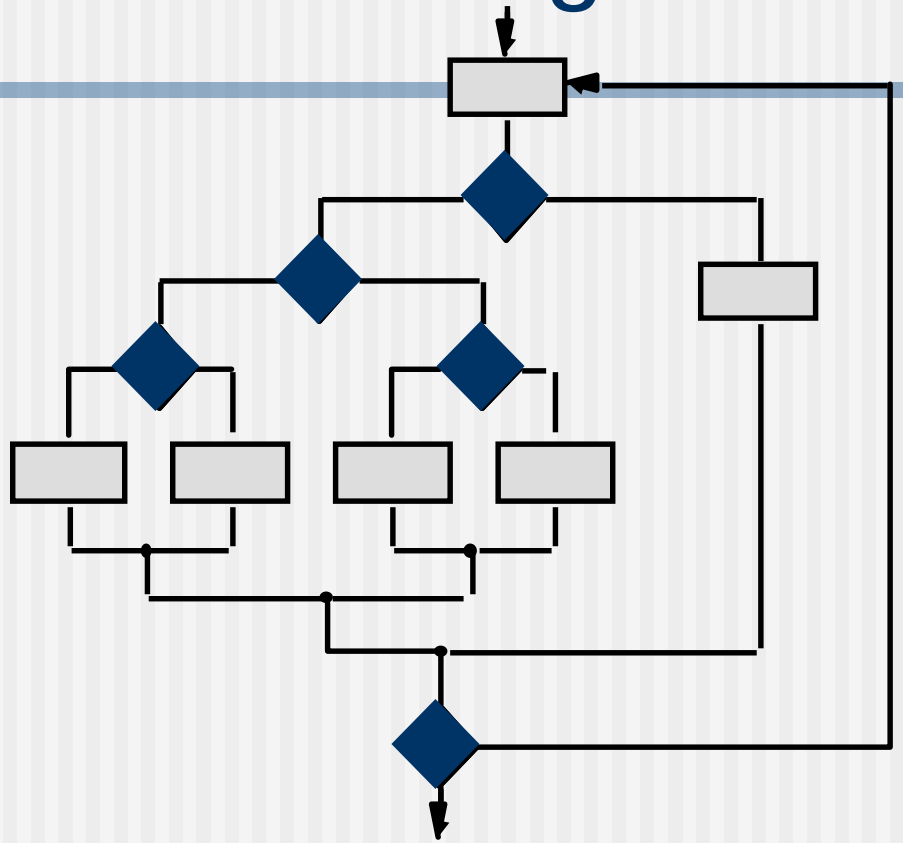


OBJECTIVE **to uncover errors**

CRITERIA **in a complete manner**

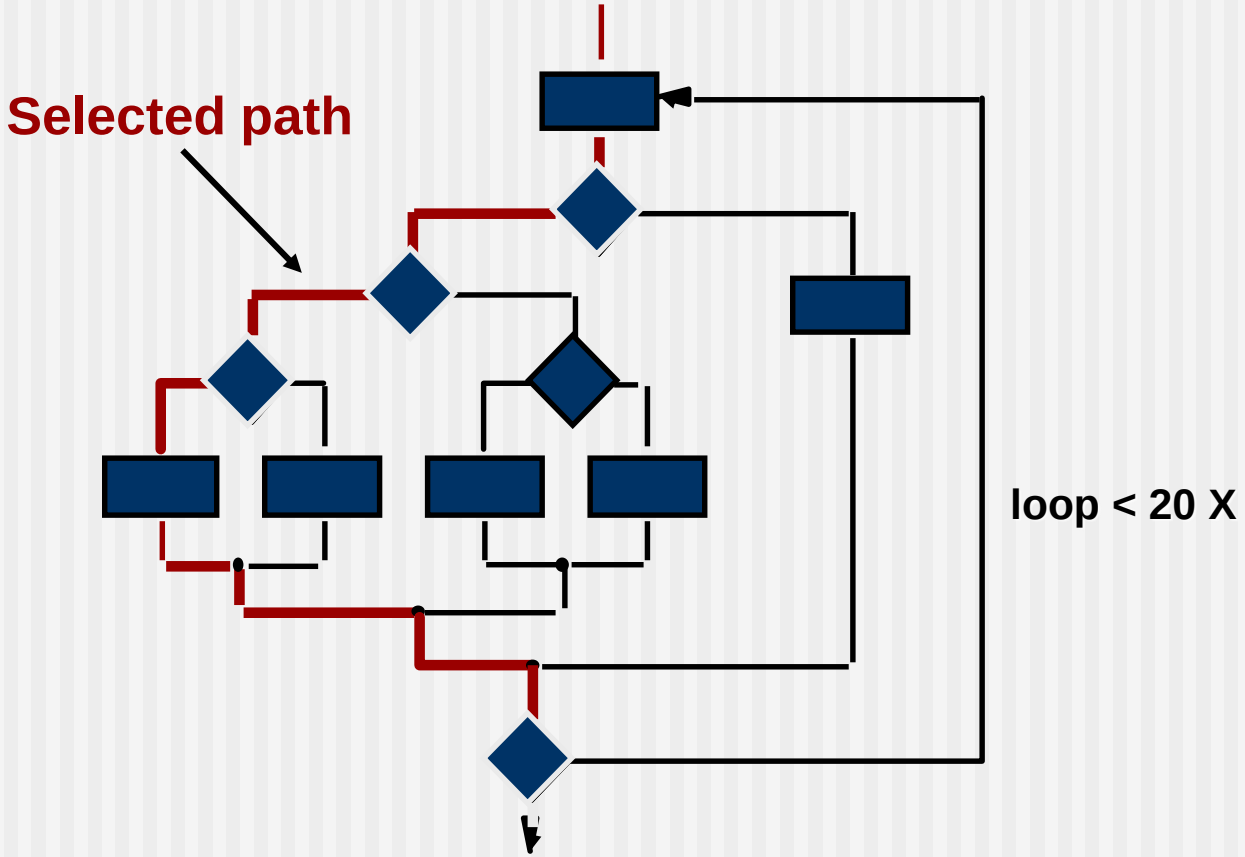
CONSTRAINT **with a minimum of effort and time**

Exhaustive Testing

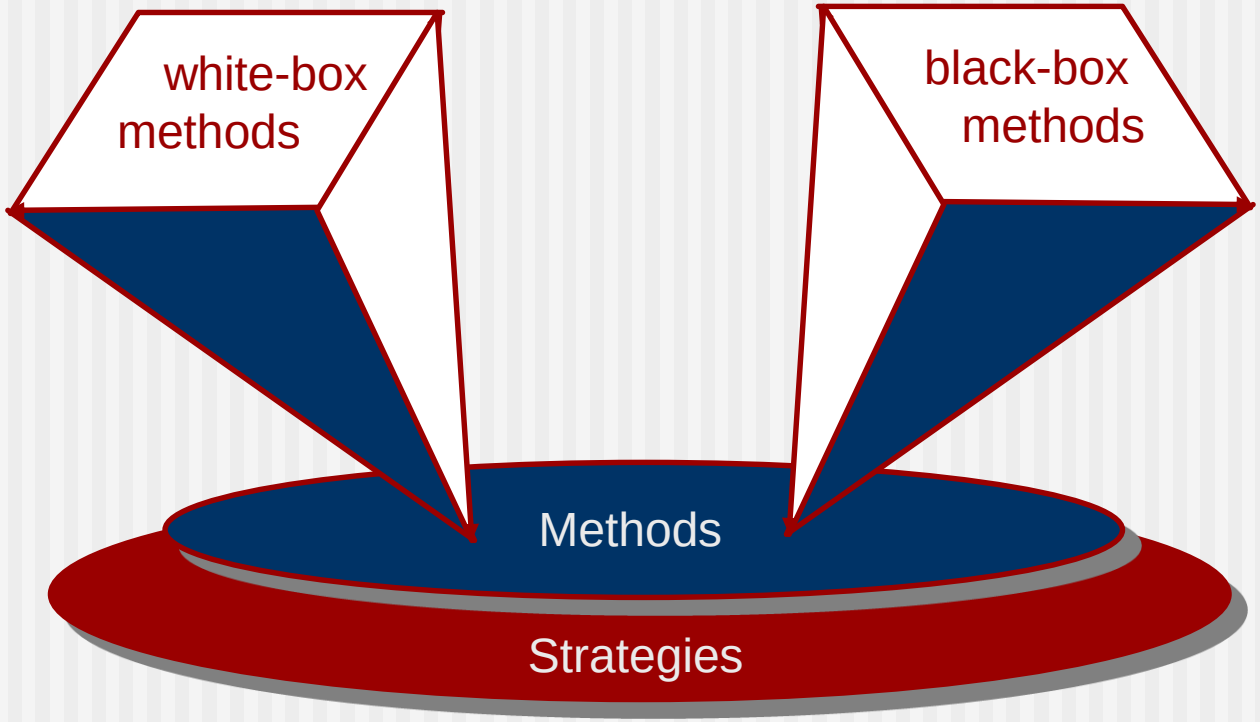


There are 10^{14} possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!

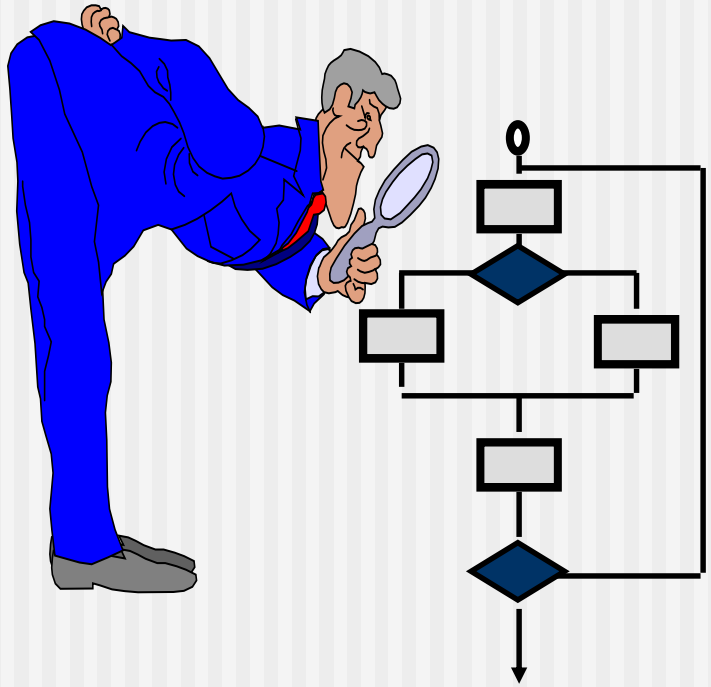
Selective Testing



Software Testing



White-Box Testing

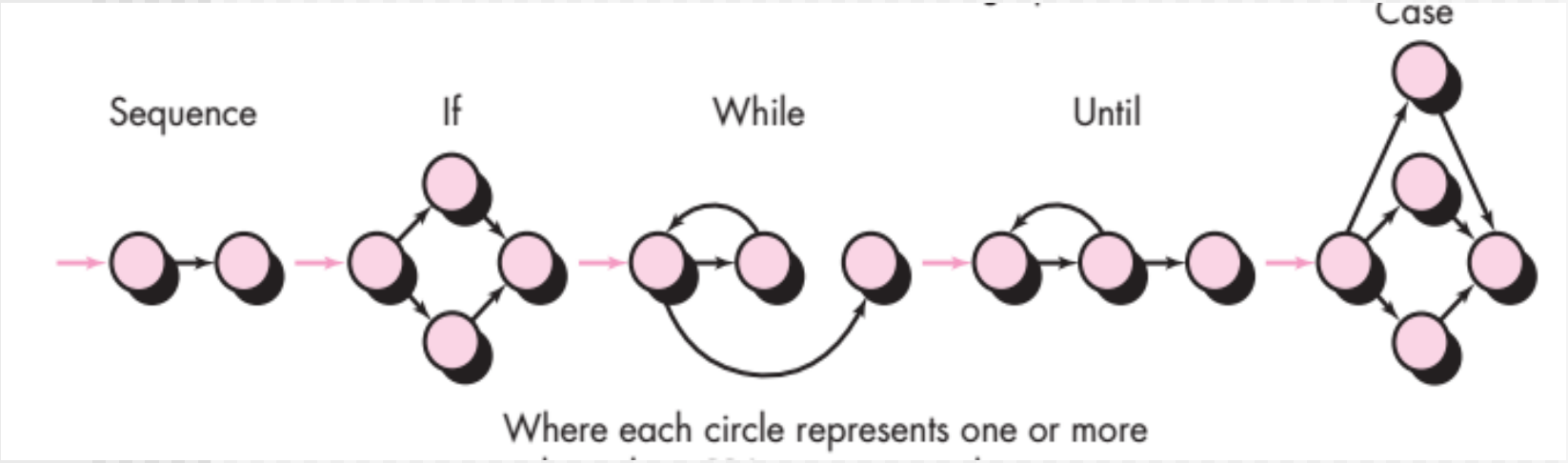


... our goal is to ensure that all statements and conditions have been executed at least once ...

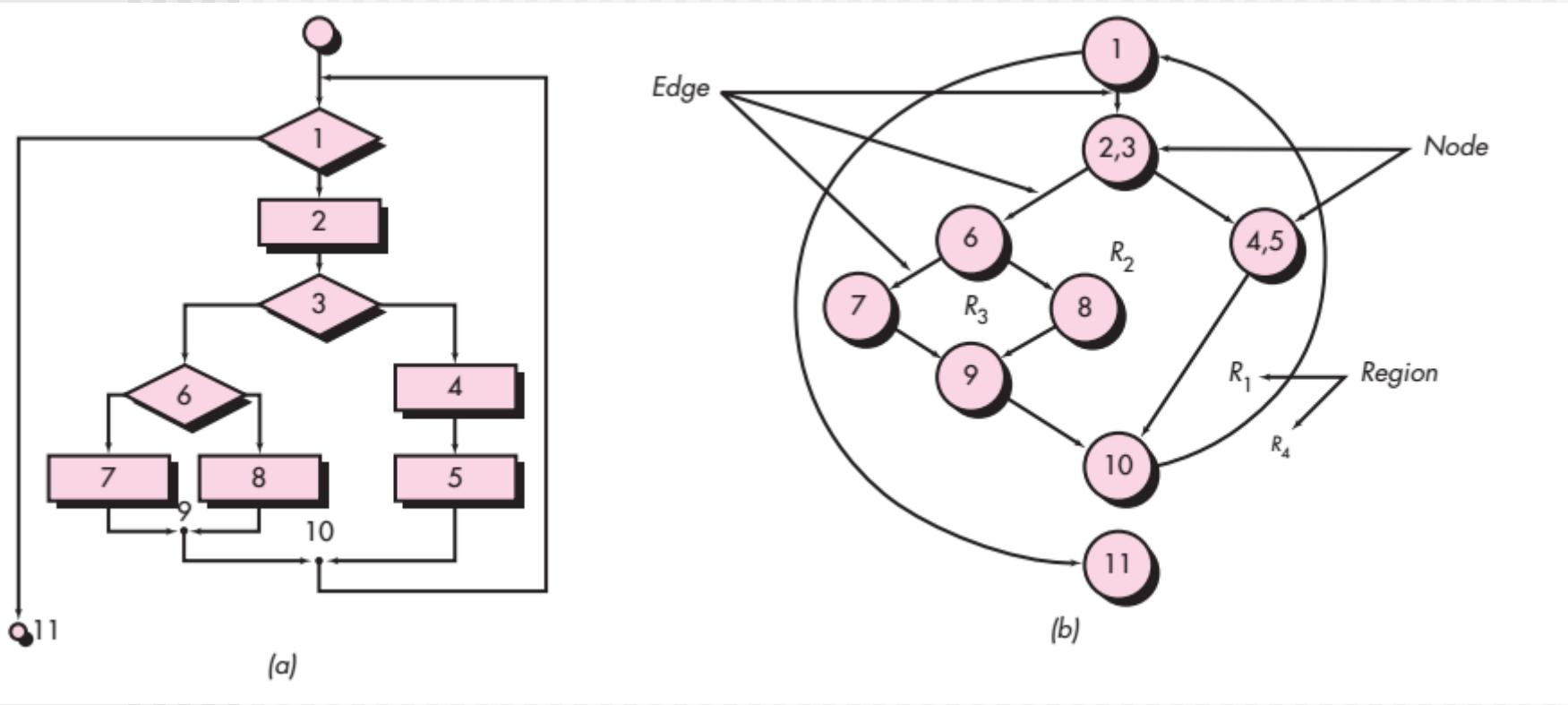
Why Cover?

- **logic errors and incorrect assumptions are inversely proportional to a path's execution probability**
- **we often believe 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**

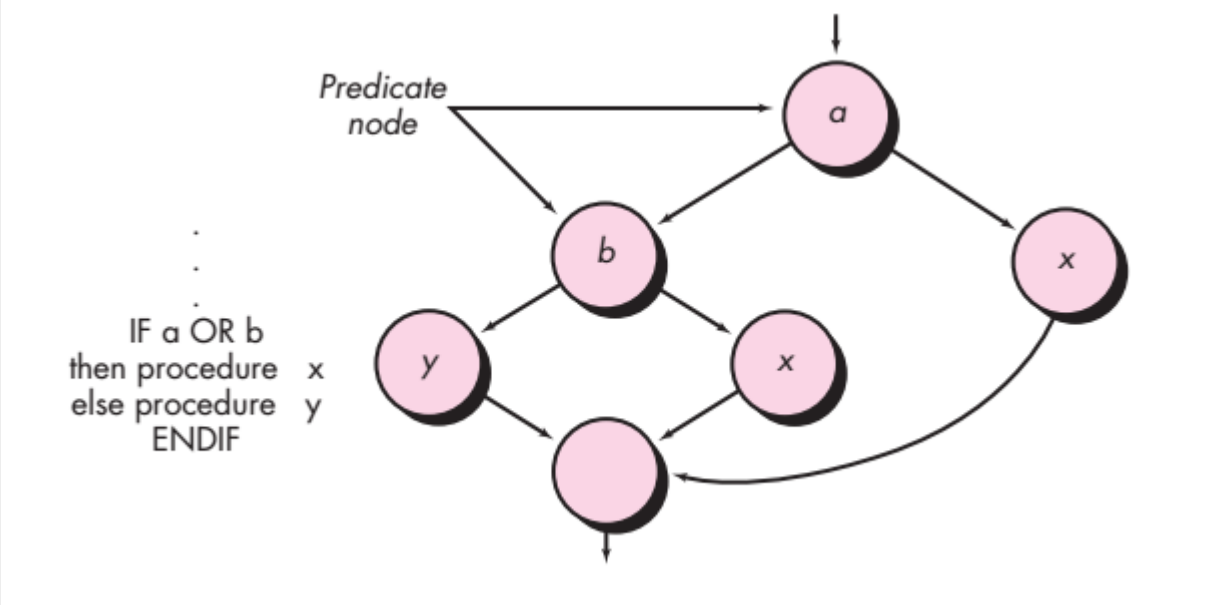
Flow Graph


Sequence
If
While
Until
Case
Where each circle represents one or more

Flow Graph



Flow Graph


Predicate node
a
b
x
y
x
IF a OR b
then procedure x
else procedure y
ENDIF

Flow Graph

مسیر مستقل

Path 1: 1-11

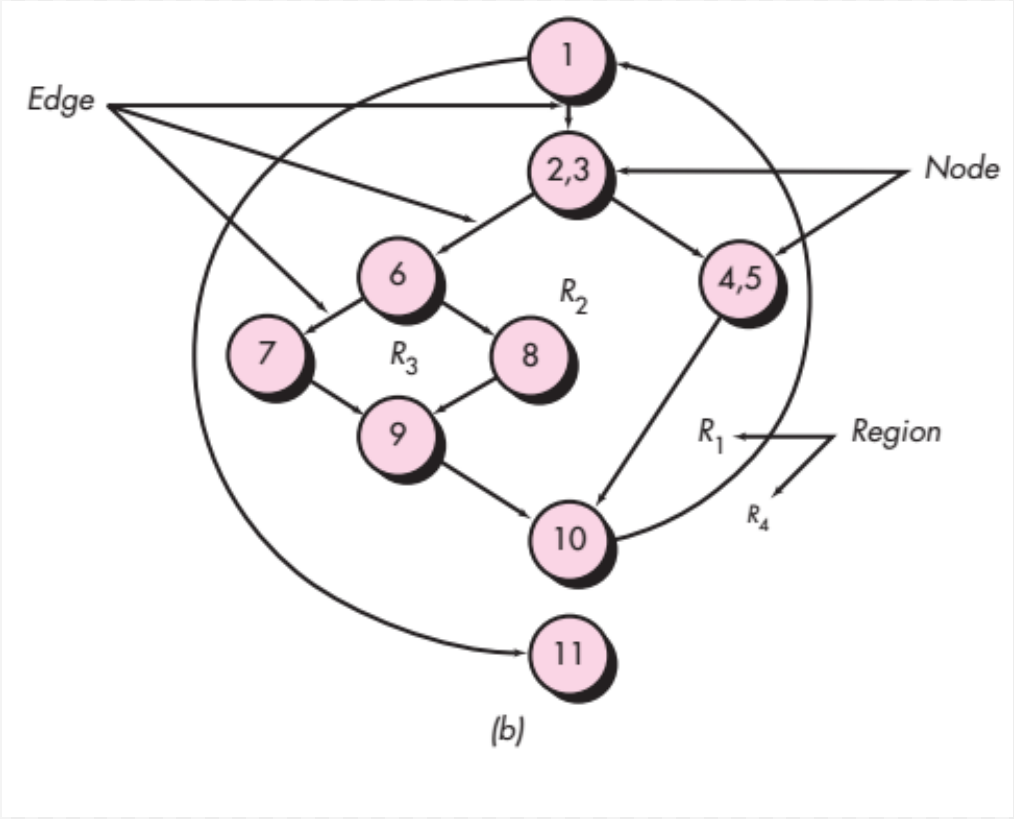
Path 2: 1-2-3-4-5-10-1-11

Path 3: 1-2-3-6-8-9-10-1-11

Path 4: 1-2-3-6-7-9-10-1-11

مسیر غیر مستقل

1-2-3-4-5-10-1-2-3-6-8-9-10-1-11



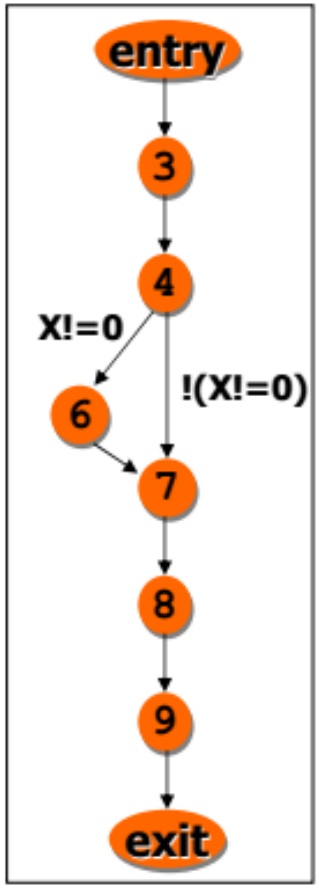
Statement coverage

Test requirements: Statements in program

$$C_{stmts} = \frac{\text{(number of executed statements)}}{\text{(number of statements)}}$$

Statement coverage: Example

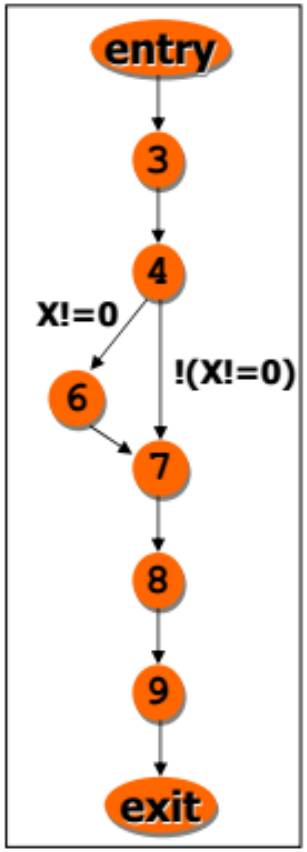
```
1. void main() {
2.     float x, y;
3.     read(x);
4.     read(y);
5.     if (x!=0)
6.         x = x+10;
7.     y = y/x;
8.     write(x);
9.     write(y);
10. }
```



Identify test cases for statement coverage

Statement coverage: Example

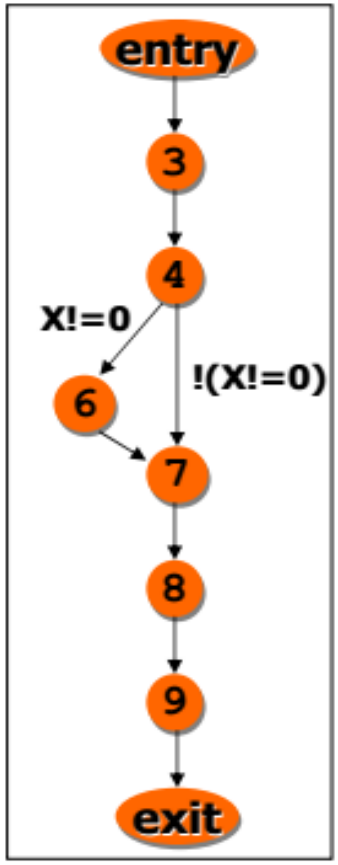
```
1. void main() {
2.     float x, y;
3.     read(x);
4.     read(y);
5.     if (x!=0)
6.         x = x+10;
7.     y = y/x;
8.     write(x);
9.     write(y);
10. }
```



- Test requirements
  - Nodes 3, …, 9
- Test specification
  - (x!=0, any y)
- Test cases
  - (x=20, y=30)
- Such test does not reveal the fault at statement 7
- To reveal it, we need to traverse edge 4-7
- ⇒ *branch coverage*

Branch coverage: Example

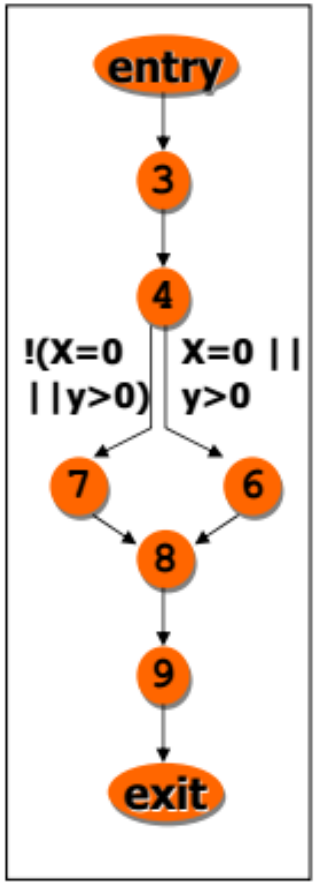
```
1. void main() {
2.     float x, y;
3.     read(x);
4.     read(y);
5.     if (x!=0)
6.       x = x+10;
7.     y = y/x;
8.     write(x);
9.     write(y);
10. }
```



- Test requirements
  - Edges 4-6 and 4-7
- Test specification
  - (x!=0, any y)
  - (x=0, any y)
- Test cases
  - (x=20, y=30)
  - (x=0, y=30)}

Branch coverage: Example

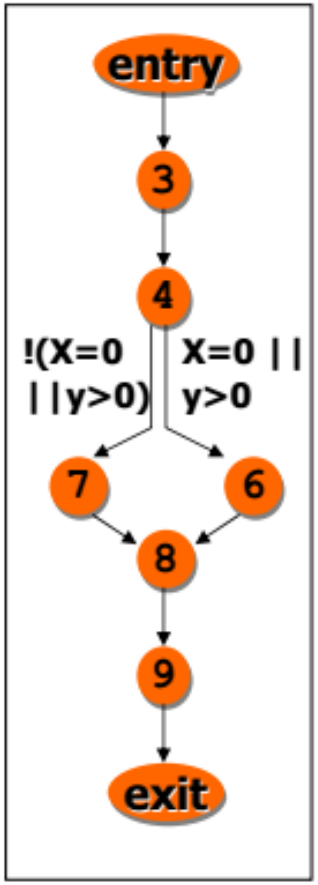
```
1. void main() {
2.   float x, y;
3.   read(x);
4.   read(y);
5.   if(x==0)||(y>0)
6.     y = y/x;
7.   else x = y+2;
8.   write(x);
9.   write(y);
10.}
```



- Consider test cases {(x=5,y=5), (x=5, y=-5)}

Branch coverage: Example

```
1. void main() {
2.   float x, y;
3.   read(x);
4.   read(y);
5.   if(x==0)||(y>0)
6.     y = y/x;
7.   else x = y+2/x;
8.   write(x);
9.   write(y);
10.}
```



- Consider test cases {(x=5,y=5), (x=5, y=-5)}
- The test suite is adequate for branch coverage, but does not reveal the fault at statement 6
- Predicate 4 can be true or false operating on only one condition

⇒ Basic *condition coverage*

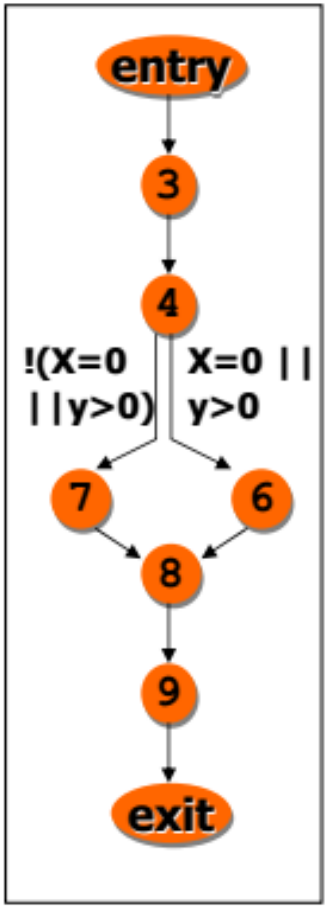
Condition Coverage

Test requirements: Truth values assumed by basic conditions

$$C_{bc} = \frac{\text{(number of boolean values assumed by all basic conditions)}}{\text{(number of boolean values of all basic conditions)}}$$

Condition Coverage : Example

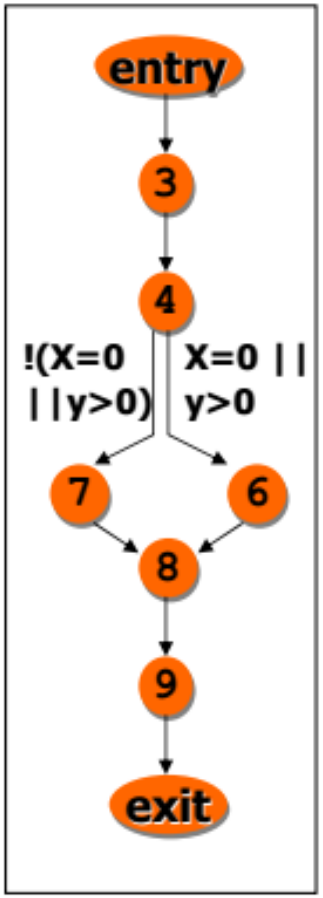
```
1. void main() {
2.   float x, y;
3.   read(x);
4.   read(y);
5.   if(x==0)||(y>0)
6.     y = y/x;
7.   else x = y+2;
8.   write(x);
9.   write(y);
10.}
```



- Consider test cases {(x=0,y=-5), (x=5, y=5)}

Condition Coverage : Example

```
1. void main() {
2.   float x, y;
3.   read(x);
4.   read(y);
5.   if(x==0)||(y>0)
6.     y = y/x;
7.   else x = y+2;
8.   write(x);
9.   write(y);
10.}
```



- Consider test cases {(x=0,y=-5), (x=5, y=5)}
- The test suite is adequate for basic condition coverage, but it does not reveal the fault at statement 6
- The test suite is not adequate for branch coverage.

⇒*Branch and condition coverage*

Branch and Condition Coverage

Test requirements: Branches and truth values assumed by basic conditions

if ((a || b) && c) { … }

a	b	c	Outcome
T	T	T	T
F	F	F	F

Full predicate coverage (FPC)

- A test set achieves FPC when each condition in the program is forced to true and to false in a scenario where that condition is *directly correlated* with the outcome of the decision. A condition *c* is directly correlated with its decision *d* when either $d \Leftrightarrow c$ holds or $d \Leftrightarrow not(c)$ holds
- For a decision containing *N* conditions, a maximum of 2*N* tests are required to achieve FPC

Modified condition/decision coverage (MC/DC)

- This strengthens the *directly correlated* requirement of FPC by requiring the condition *c* to *independently affect* the outcome of the decision *d*.
- A condition is shown to independently affect a decision's outcome by varying just that condition while holding fixed all other possible conditions
- For a decision containing *N* conditions, a maximum of 2*N* tests are required to achieve MC/DC (the same upper bound as FPC), so the number of tests is proportional to the complexity of the decision (the number of conditions within it)

Multiple condition coverage (MCC)

((((a || b) && c) || d) && e)

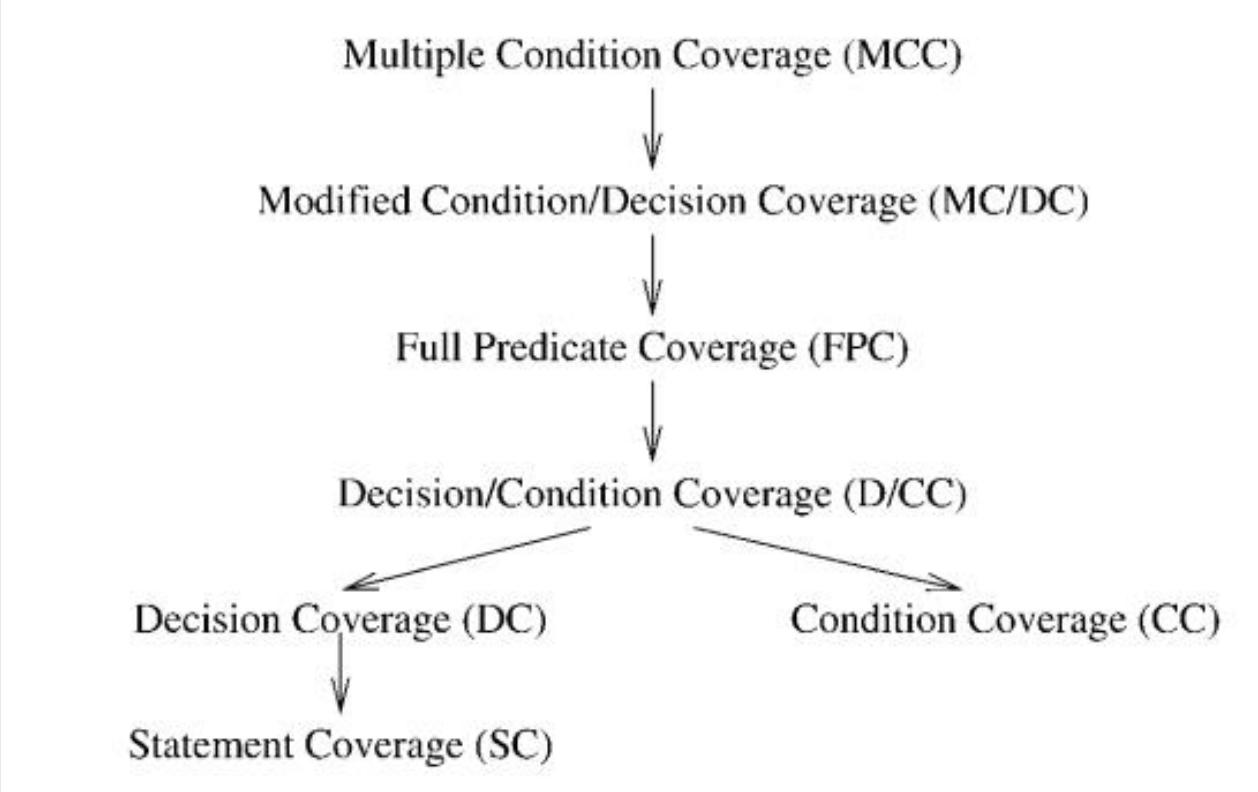
Test case	a	b	c	d	e
1	True	-	True	-	True
2	False	True	True	-	True
3	True	-	False	True	True
4	False	True	False	True	True
5	False	False	-	True	True
6	True	-	True	-	False
7	False	True	True	-	False
8	True	-	False	True	False
9	False	True	False	True	False
10	False	False	-	True	False
11	True	-	False	False	-
12	False	True	False	False	-
13	False	False	-	False	-

Multiple condition coverage (MCC)

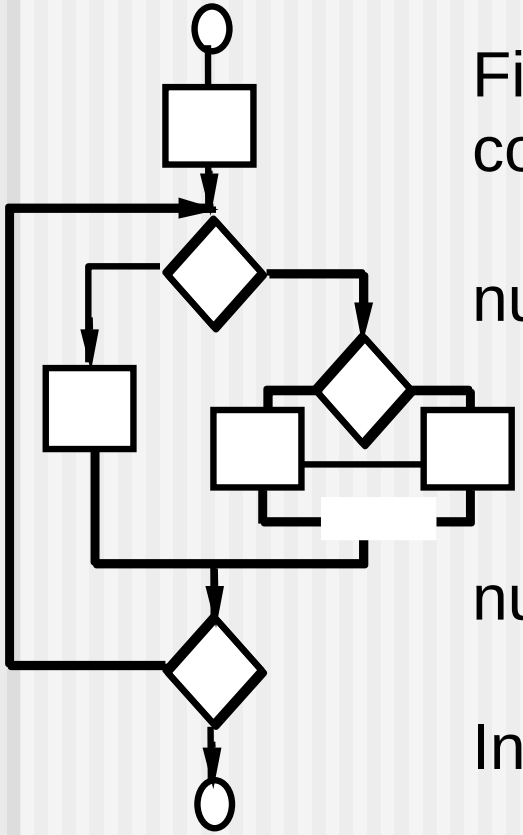
((((a || b) && c) || d) && e)

Test case	a	b	c	d	e
1	True	-	True	-	True
2	False	True	True	-	True
3	True	-	False	True	True
4	False	True	False	True	True
5	False	False	-	True	True
6	True	-	True	-	False
7	False	True	True	-	False
8	True	-	False	True	False
9	False	True	False	True	False
10	False	False	-	True	False
11	True	-	False	False	-
12	False	True	False	False	-
13	False	False	-	False	-

The hierarchy of control-flow coverage



Basis Path Testing



First, we compute the cyclomatic complexity:

number of simple decisions + 1

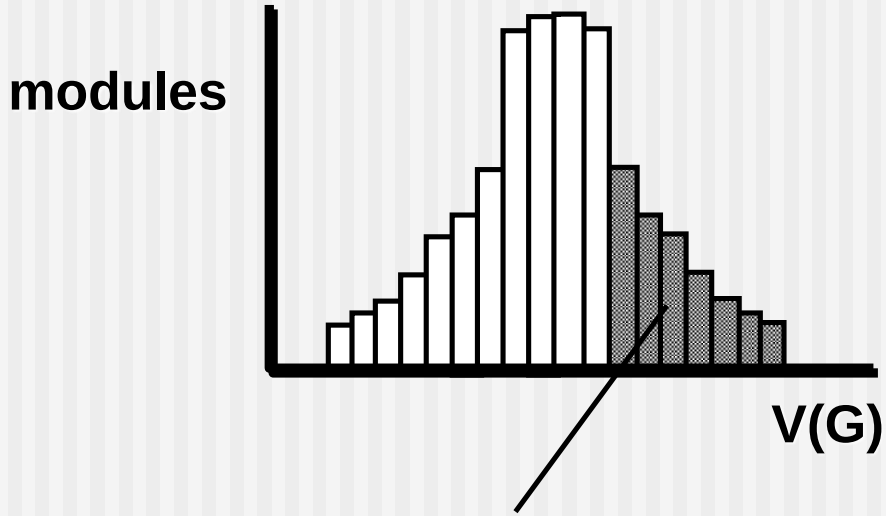
or

number of enclosed areas + 1

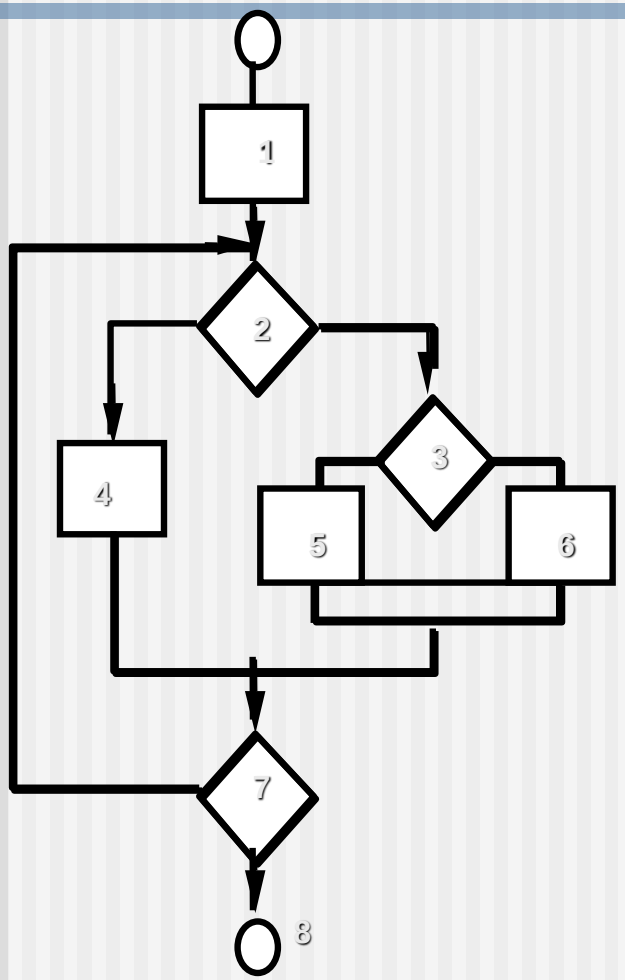
In this case, V(G) = 4

Cyclomatic Complexity

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



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

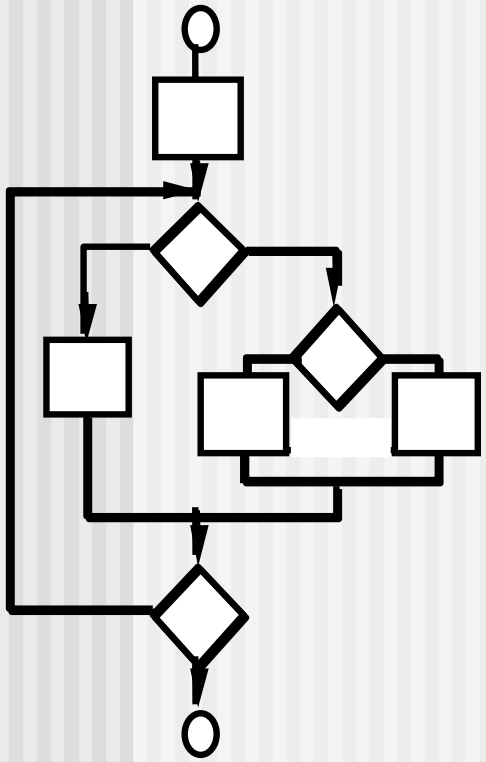
Path 2: 1,2,3,5,7,8

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

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.

Deriving Test Cases

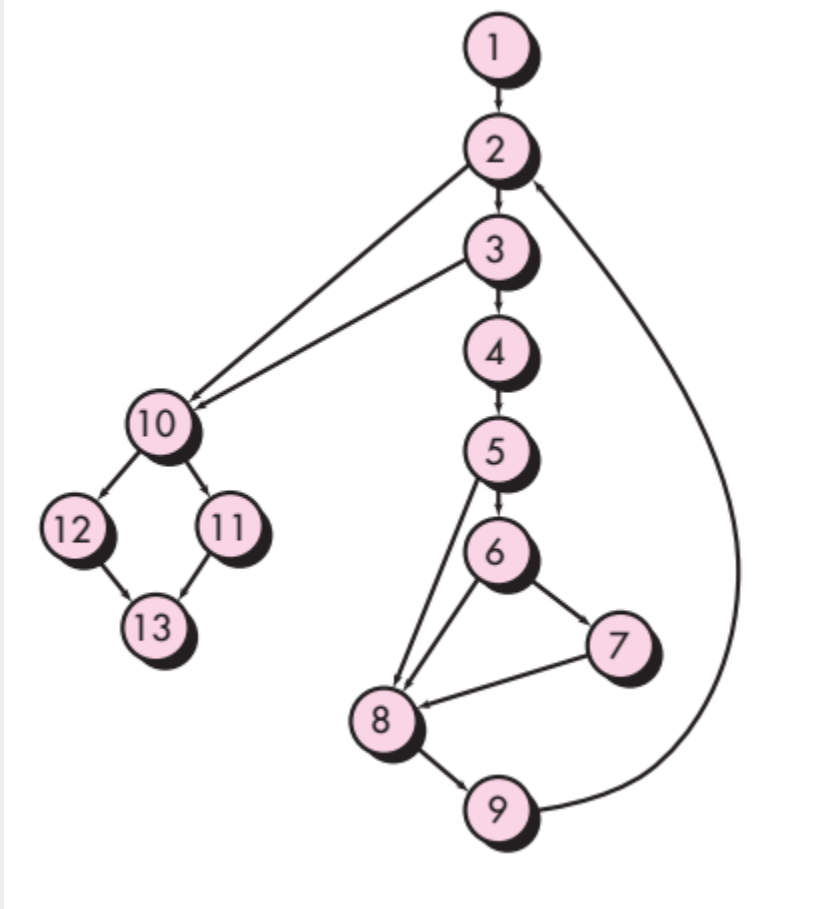
- *Summarizing:*
  - Using the design or code as a foundation, draw a corresponding flow graph.
  - Determine the cyclomatic complexity of the resultant flow graph.
  - Determine a basis set of linearly independent paths.
  - Prepare test cases that will force execution of each path in the basis set.

Deriving Test Cases

```
* 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
```

Deriving Test Cases

```
*  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;
1  i = 1;
   total.input = total.valid = 0;          2
   sum = 0;
   DO WHILE value[i] <> -999 AND total.input < 100   3
4     increment total.input by 1;
      IF value[i] > = minimum AND value[i] < = maximum   6
5        THEN increment total.valid by 1;
7                sum = s sum + value[i]
         ELSE skip
8     ENDIF
      increment i by 1;
9  ENDDO
   IF total.valid > 0   10
11    THEN average = sum / total.valid;
12    ELSE average = -999;
13 ENDIF
END average
```



Deriving Test Cases

$V(G)$ = 6 regions
$V(G)$ = 17 edges − 13 nodes + 2 = 6
$V(G)$ = 5 predicate nodes + 1 = 6

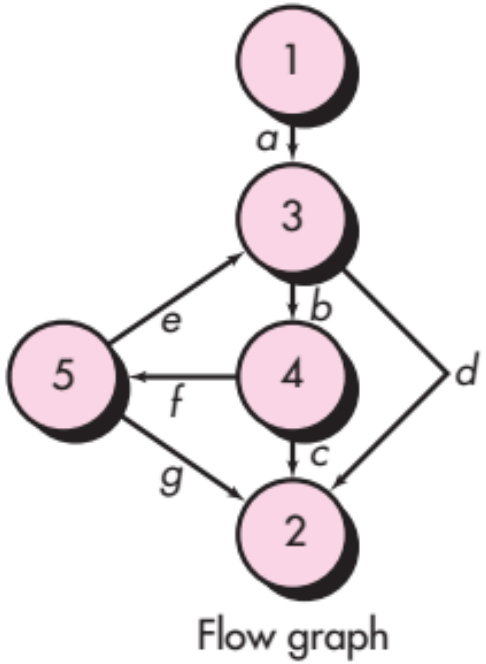
Path 1: 1-2-10-11-13
Path 2: 1-2-10-12-13

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-. . .

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

Graph Matrices



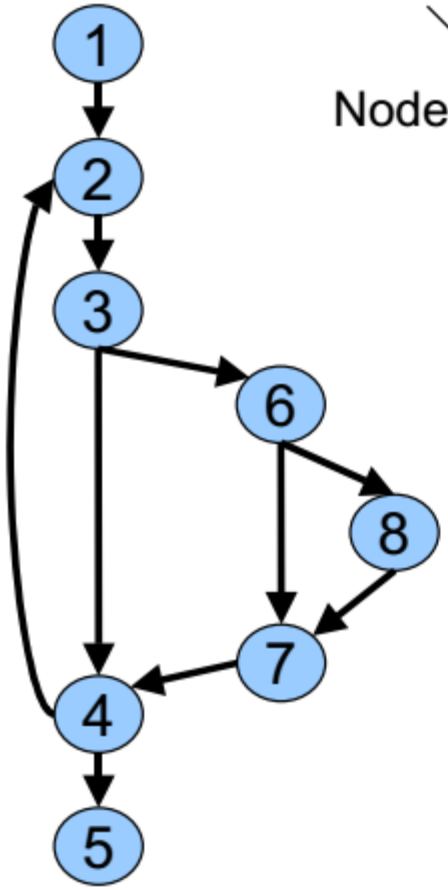
Flow graph

Node \ Connected to node	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph matrix

Graph Matrices

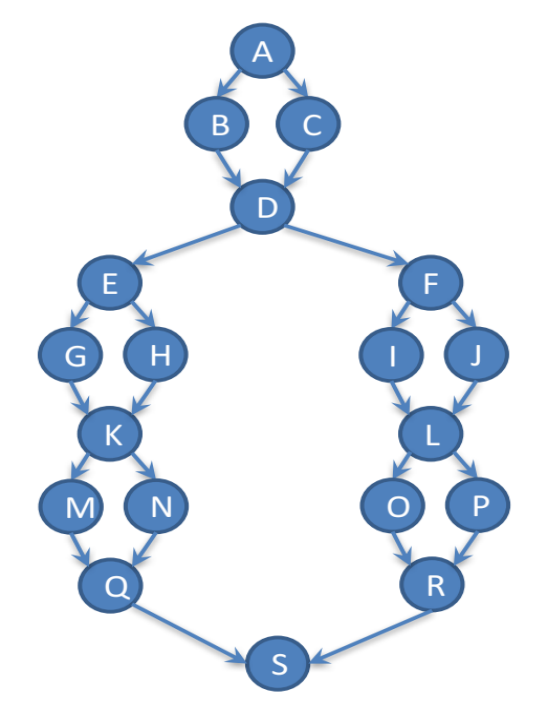
The Graph (Connection) Matrix



Node	Connected to node 1	2	3	4	5	6	7	8	
1	0	1	0	0	0	0	0	0	1 – 1 = 0
2	0	0	1	0	0	0	0	0	1 – 1 = 0
3	0	0	0	1	0	1	0	0	2 - 1 = 1
4	0	1	0	0	1	0	0	0	2 – 1 = 1
5	0	0	0	0	0	0	0	0	---
6	0	0	0	0	0	0	1	1	2 – 1 = 1
7	0	0	0	1	0	0	0	0	1 – 1 = 0
8	0	0	0	0	0	0	1	0	1 – 1 = 0

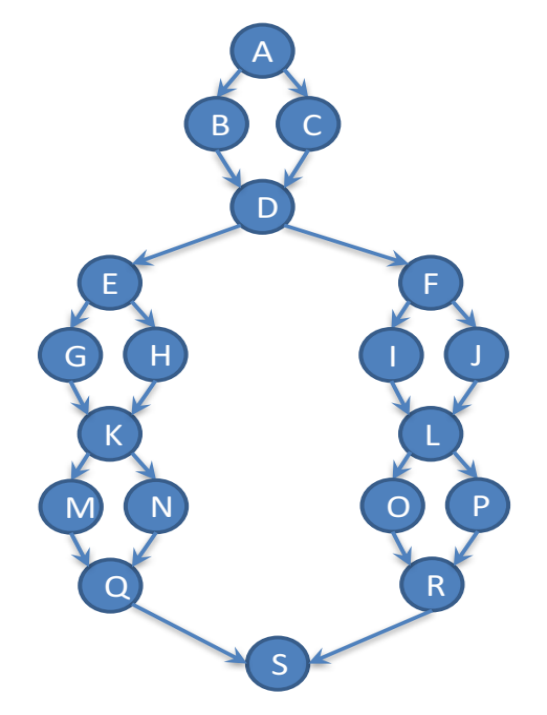
3+1=4

Basis path testing example



- V(G) = 7 regions
- V(G) = 6 predicate node +1=7
- V(G)= 24-19 +2=7

Basis path testing



7 independent paths:

1. ABDEGKMQS
2. ACDEGKMQS
3. ABDFILORS
4. ABDEHKMQS
5. ABDEGKNQS
6. ACDFJLORS
7. ACDFILPRS

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

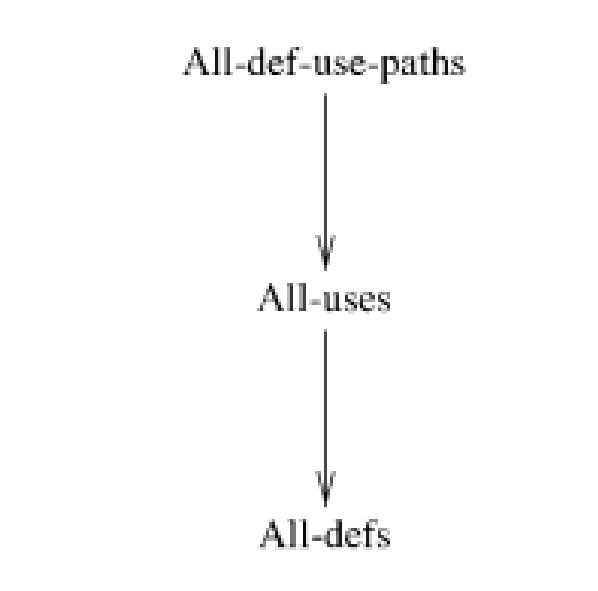
Data Flow Testing

- 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'*

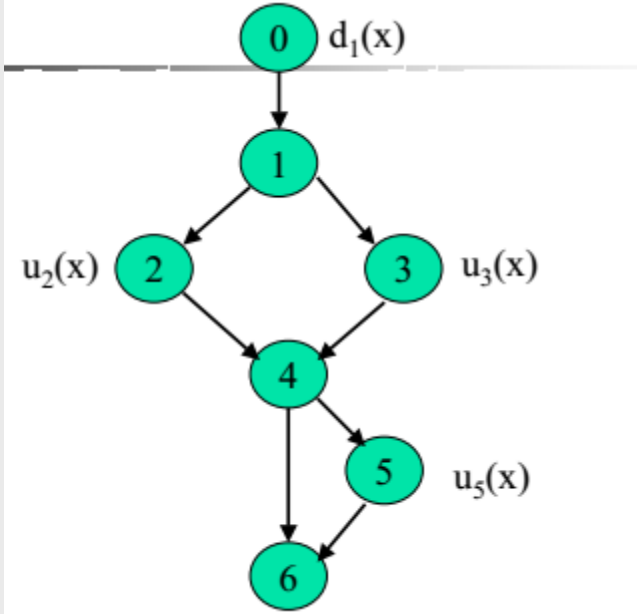
Data Flow Testing

- **All-defs:**
  - The *all-definitions* criterion requires a test suite to test at least one def-use pair (v,d_v, u_v) for every definition *dv*, that is, at least one path from each definition to one of its feasible uses.
- **All-uses**
  - The *all-uses* criterion requires a test suite to test all def-use pairs (v,d_v, u_v). This means testing all feasible uses of all definitions
- **All-def-use-paths:**
  - The *all-def-use-paths* criterion requires a test suite to test all def-use pairs (v,d_v, u_v).and to test all paths from *dv* to *uv*. This is the strongest data-flow criterion and usually requires unrealistically many test cases.

Data Flow Testing



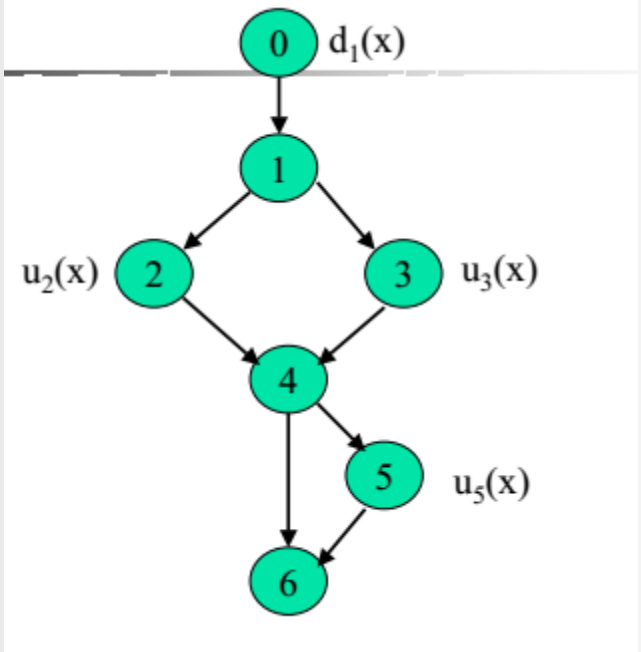
Data Flow Testing



All-Defs

- Requires:
  $d_1(x)$ to a use
- Satisfactory path:
  - 0, 1, 2, 4, 6

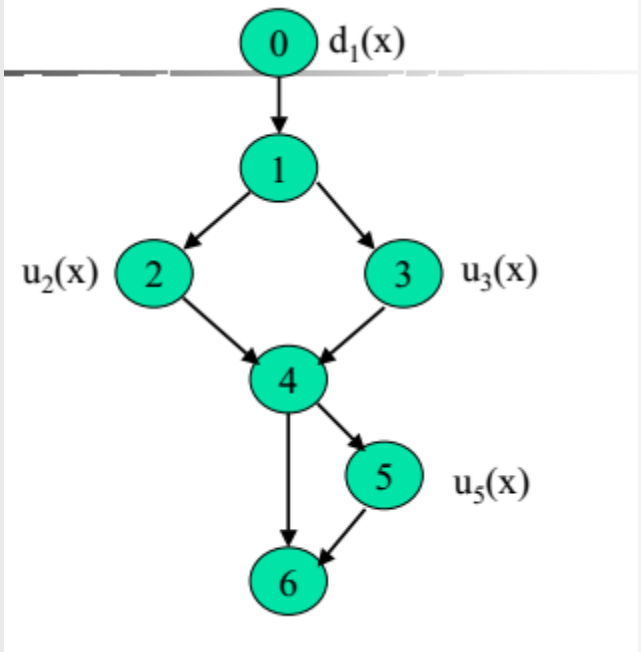
Data Flow Testing



All-Uses

- Requires:
  - $d_1(x)$ to a $u_2(x)$
  - $d_1(x)$ to a $u_3(x)$
  - $d_1(x)$ to a $u_5(x)$
- Satisfactory path:
  - 0, 1, 2, 4, 5, 6
  - 0, 1, 3, 4, 6

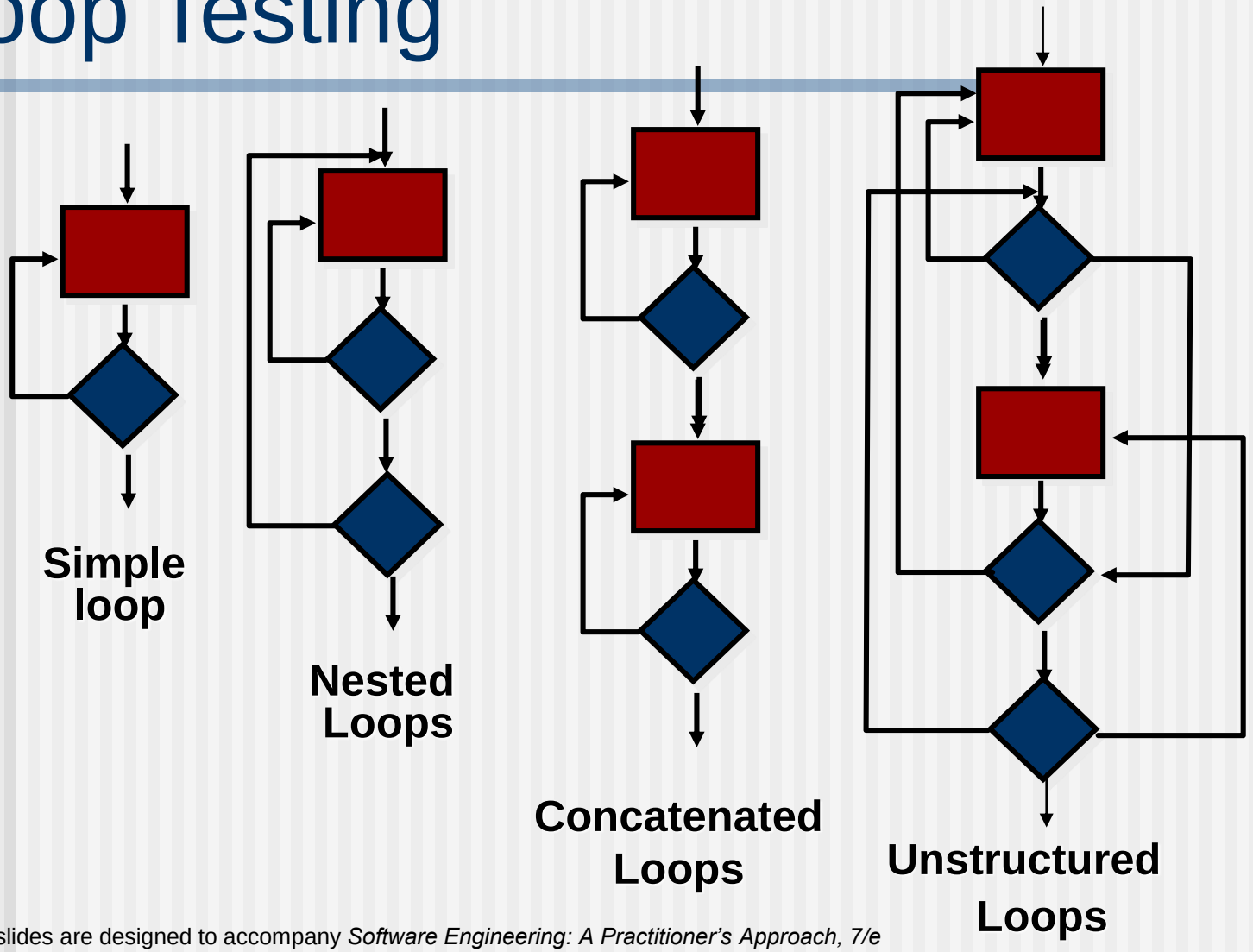
Data Flow Testing



All-DU-Paths

- Requires:
  - $d_1(x)$ to a $u_2(x)$
  - $d_1(x)$ to a $u_3(x)$
  - Both paths for $d_1(x)$ to a $u_5(x)$
- Satisfactory path:
  - 0, 1, 2, 4, 5, 6
  - 0, 1, 3, 4, 5, 6

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.

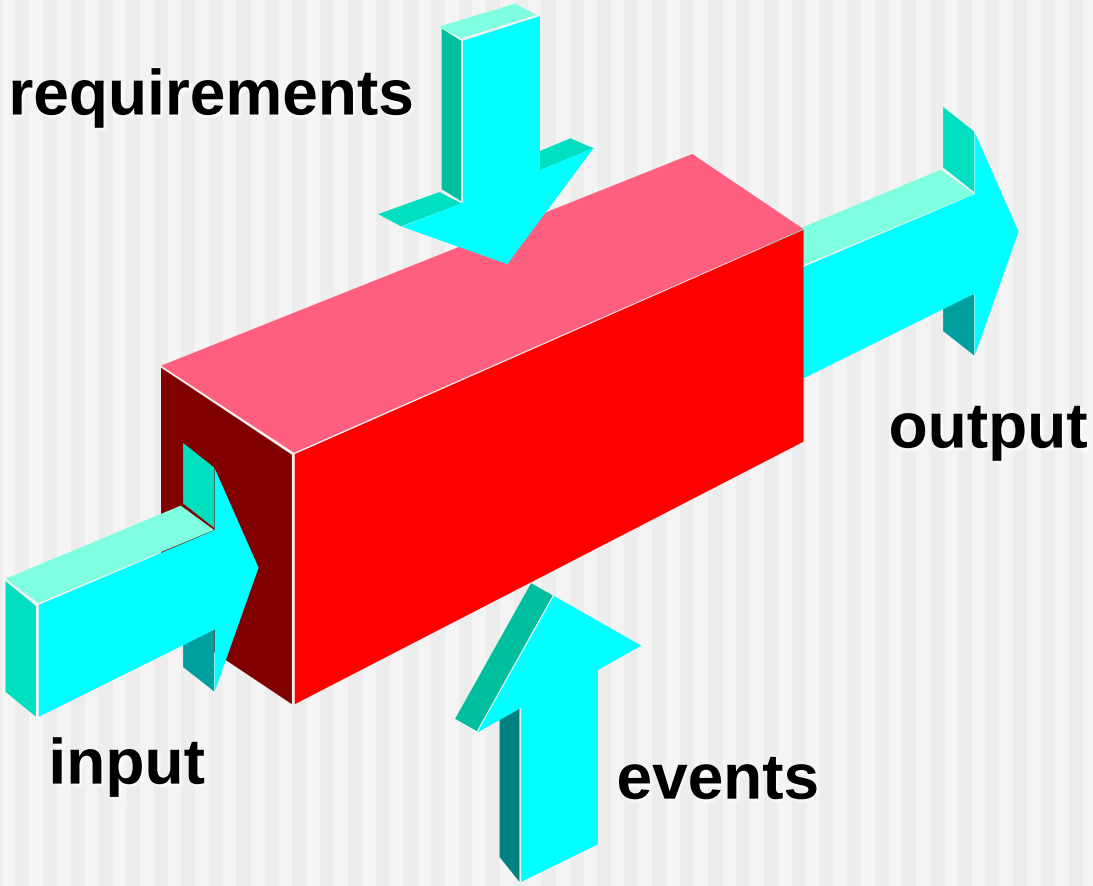
Move out one loop and set it up as in step 2, holding all other loops at typical values. Continue this step until the outermost loop has been tested.

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.

Black-Box Testing



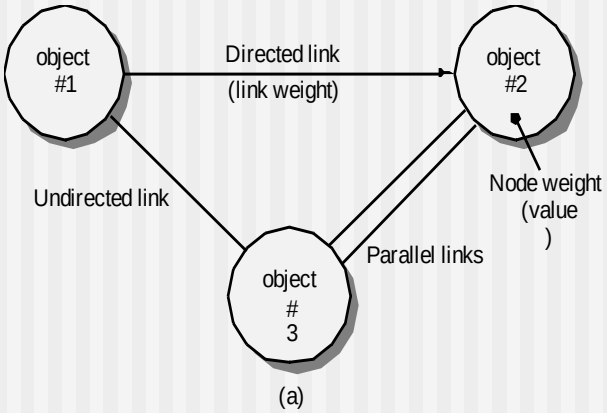
Black-Box Testing

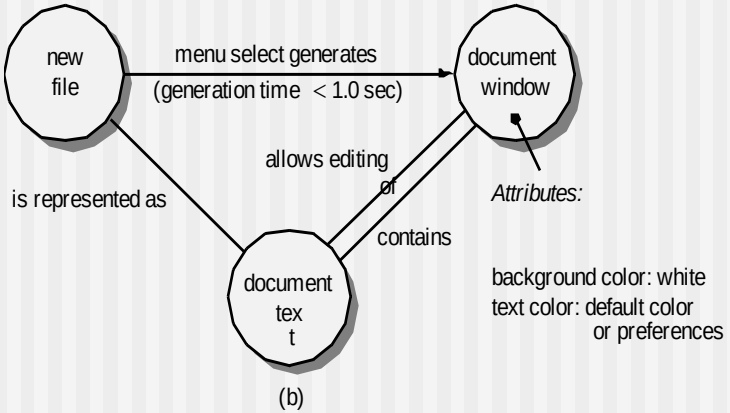
- How is functional validity tested?
- How is system behavior and performance tested?
- What classes of input will make good test cases?
- Is the system particularly sensitive to certain input values?
- How are the boundaries of a data class isolated?
- What data rates and data volume can the system tolerate?
- What effect will specific combinations of data have on system operation?

Graph-Based Methods

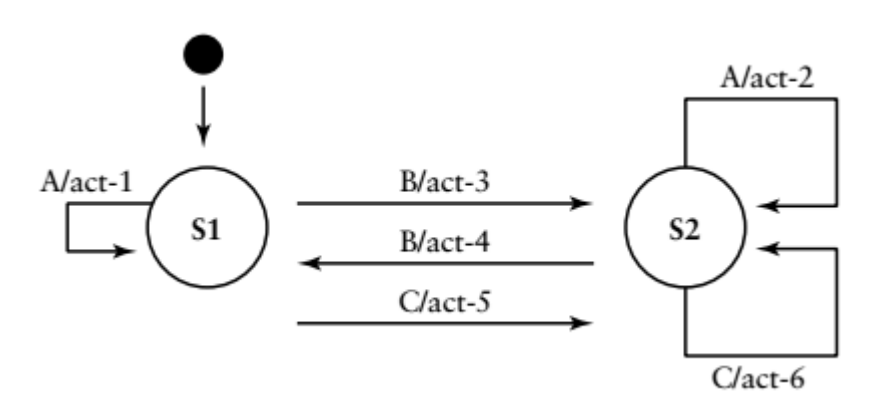
To understand the objects that are modeled in software and the relationships that connect these objects

In this context, we consider the term "objects" in the broadest possible context. It encompasses data objects, traditional components (modules), and object-oriented elements of computer software.



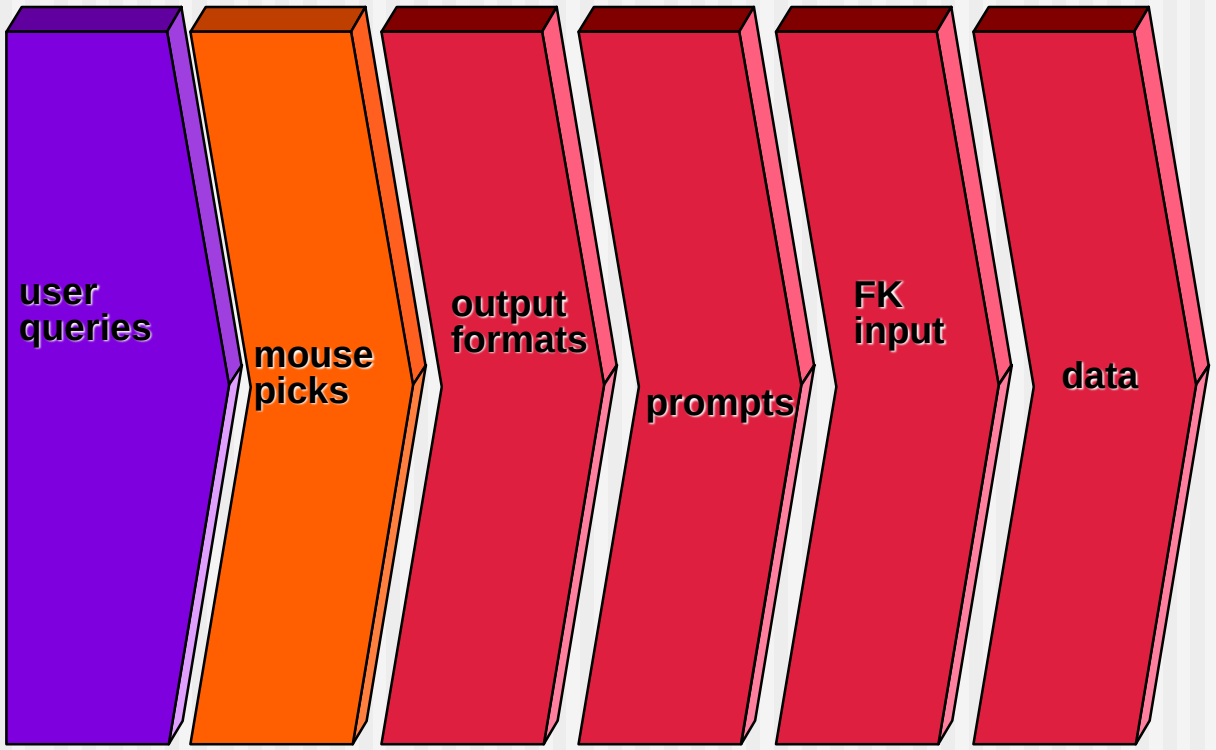


Graph-Based Methods

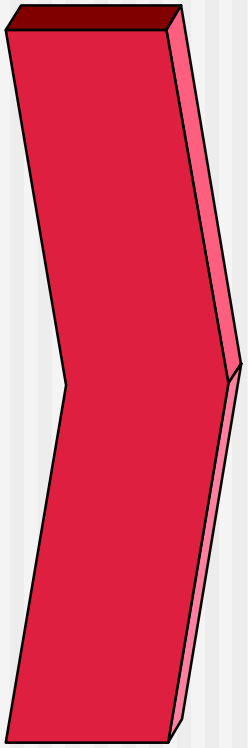


Input A in S1
Input A in S2
Input B in S1
Input B in S2
Input C in S1
Input C in S2

Equivalence Partitioning



Sample Equivalence Classes



Valid data

- **user supplied commands**
- **responses to system prompts**
- **file names**
- **computational data**
  - **physical parameters**
  - **bounding values**
  - **initiation values**
- **output data formatting**
- **responses to error messages**
- **graphical data (e.g., mouse picks)**

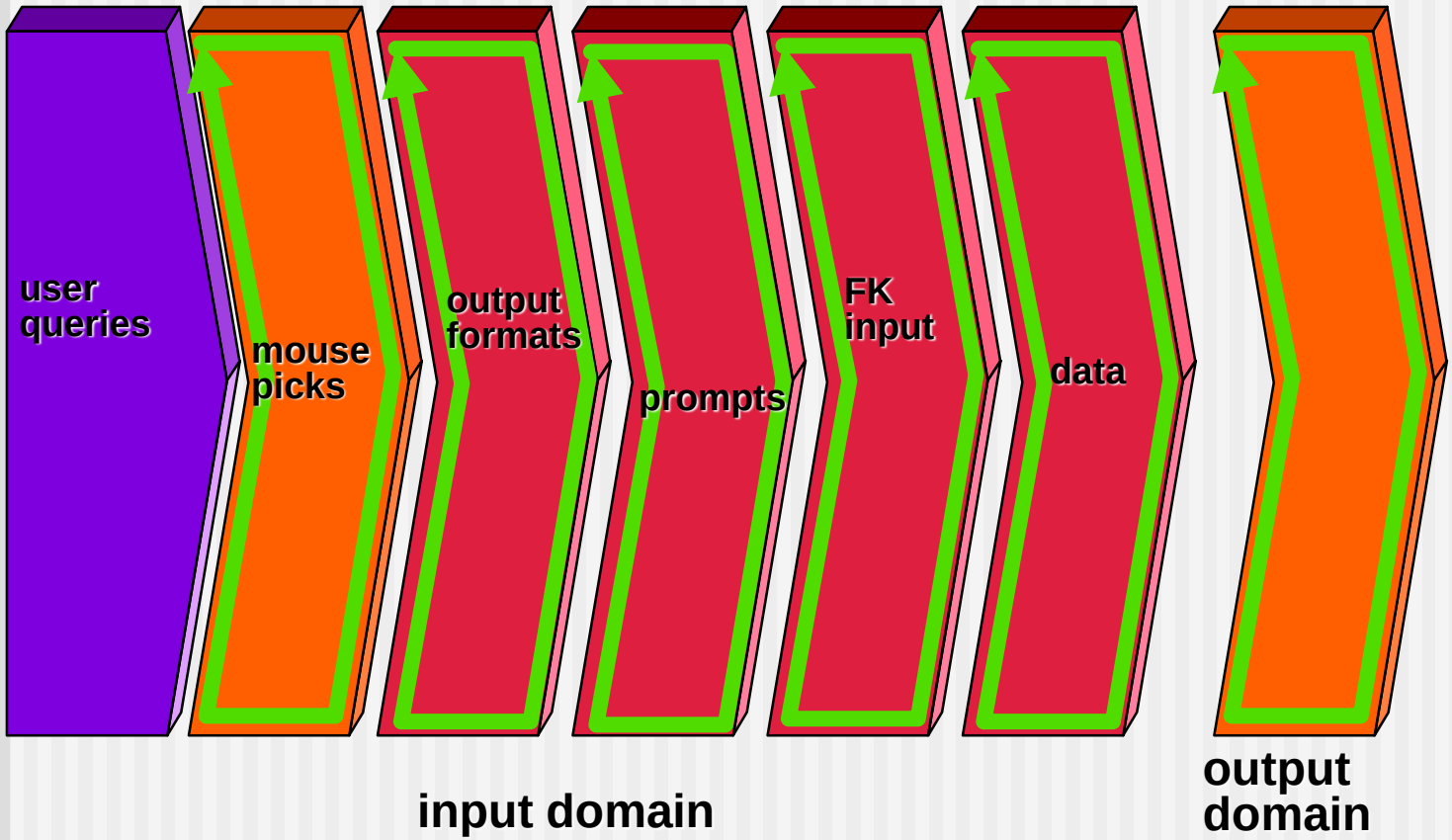
Invalid data

- **data outside bounds of the program**
- **physically impossible data**
- **proper value supplied in wrong place**

Equivalence Classes

- 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.

Boundary Value Analysis



Boundary Value Analysis

1. If an input condition specifies a range bounded by values *a* and *b,* test cases should be designed with values *a* and *b* and just above and just below *a* and *b.*
2. 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.
3. Apply guidelines 1 and 2 to output conditions.
4. If internal program data structures have prescribed boundaries (e.g., a table has a defined limit of 100 entries), be certain to design a test case to exercise the data structure at its boundary.

Equivalence Classes & Boundary Value Analysis Example

- Suppose we are testing a module that allows a user to enter new widget identifiers into a widget data base. We will focus only on selecting equivalence classes and boundary values for the inputs. The input specification for the module states that a widget identifier should consist of 3–15 alphanumeric characters of which the first two must be letters. We have three separate conditions that apply to the input: **(i) it must consist of alphanumeric characters**, **(ii) the range for the total number of characters is between 3 and 15**, and, (iii) **the first two characters must be letters.**

Equivalence Classes & Boundary Value Analysis Example

- **(i) it must consist of alphanumeric characters.**
  - EC1. Part name is alphanumeric, valid.
  - EC2. Part name is not alphanumeric, invalid.
- **(ii) the range for the total number of characters is between 3 and 15**
  - EC3. The widget identifier has between 3 and 15 characters, valid.
  - EC4. The widget identifier has less than 3 characters, invalid.
  - EC5. The widget identifier has greater than 15 characters, invalid.
- (iii) **the first two characters must be letters.**
  - EC6. The first 2 characters are letters, valid.
  - EC7. The first 2 characters are not letters, invalid.

Equivalence Classes & Boundary Value Analysis Example

- Boundary Value Analysis
  - **BLB—**a value just below the lower bound
  - **LB—**the value on the lower boundary
  - **ALB—**a value just above the lower boundary
  - **BUB—**a value just below the upper bound
  - **UB—**the value on the upper bound
  - **AUB—**a value just above the upper bound

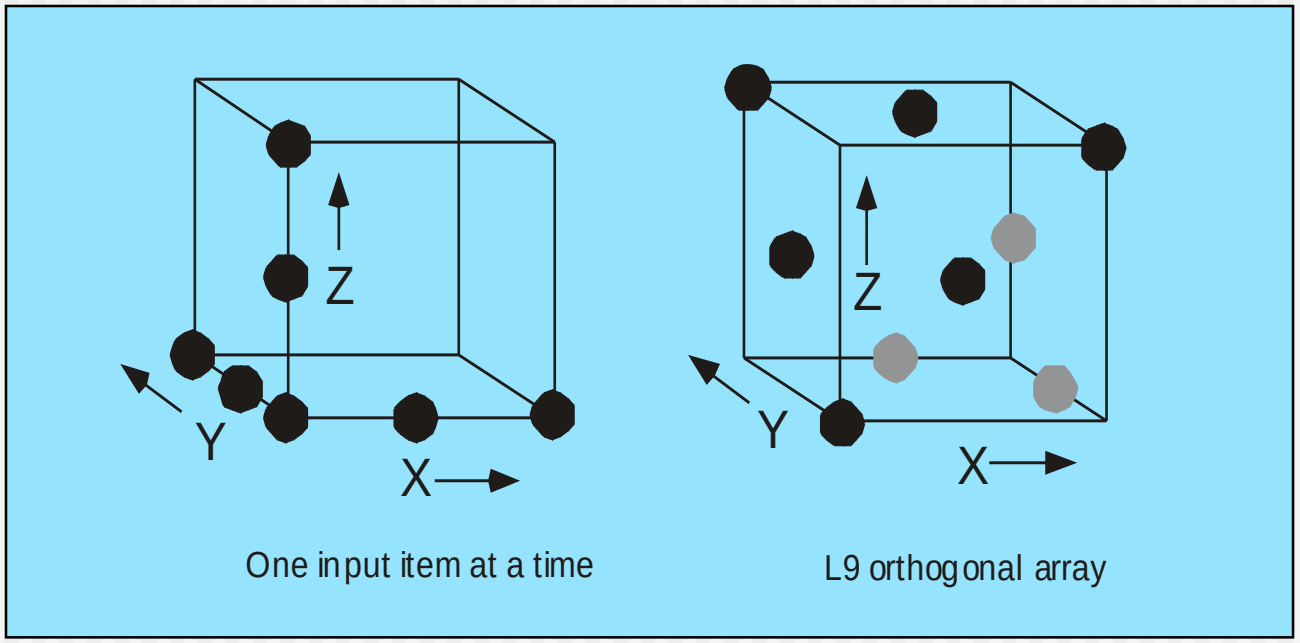
BLB—2	BUB—14
LB—3	UB—15
ALB—4	AUB—16

Equivalence Classes & Boundary Value Analysis Example

Test case identifier	Input values	Valid equivalence classes and bounds covered	Invalid equivalence classes and bounds covered
1	abc1	EC1, EC3(ALB) EC6	
2	ab1	EC1, EC3(LB), EC6	
3	abcdef123456789	EC1, EC3 (UB) EC6	
4	abcde123456789	EC1, EC3 (BUB) EC6	
5	abc*	EC3(ALB), EC6	EC2
6	ab	EC1, EC6	EC4(BLB)
7	abcdefg123456789	EC1, EC6	EC5(AUB)
8	a123	EC1, EC3 (ALB)	EC7
9	abcdef123	EC1, EC3, EC6 (typical case)	

Orthogonal Array Testing

- Used when the number of input parameters is small and the values that each of the parameters may take are clearly bounded



Model-Based Testing

- Analyze an existing behavioral model for the software or create one.
  - Recall that a *behavioral model* indicates how software will respond to external events or stimuli.
- Traverse the behavioral model and specify the inputs that will force the software to make the transition from state to state.
  - The inputs will trigger events that will cause the transition to occur.
- Review the behavioral model and note the expected outputs as the software makes the transition from state to state.
- Execute the test cases.
- Compare actual and expected results and take corrective action as required.

Software Testing Patterns

- Testing patterns are described in much the same way as design patterns (Chapter 12).
- *Example:*
  - *Pattern name:* **ScenarioTesting**
  - *Abstract:* Once unit and integration tests have been conducted, there is a need to determine whether the software will perform in a manner that satisfies users. The **ScenarioTesting** pattern describes a technique for exercising the software from the user's point of view. A failure at this level indicates that the software has failed to meet a user visible requirement. [Kan01]