# Chapter 23

## Testing Conventional Applications

*Slide Set to accompany*

*Software Engineering: A Practitioner's Approach*

**by Roger S. Pressman and Bruce R. Maxim**

# Table of Contents

# 23.1 Software Testing Fundamentals

**Testability – Capability of software to facilitate testing**

- 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
    - ☛ High test coverage
  - is not redundant.
    - ☛ Low test effort

# 23.2 Internal and External Views of Testing

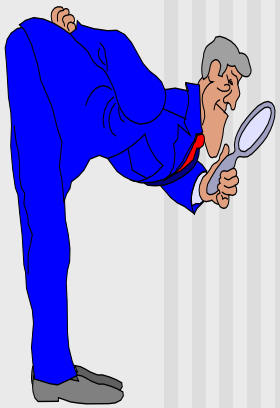
- Two ways to test any engineered product (and most other things):
  - Knowing the specified function of a product
    - Demonstrate each function is fully operational while at the same time searching for errors in each function;
  - Knowing the internal workings of a product
    - Ensure that internal operations are performed according to specifications and all internal components have been adequately exercised.

Which one is a black-box approach ?
Which one is a white-box testing?

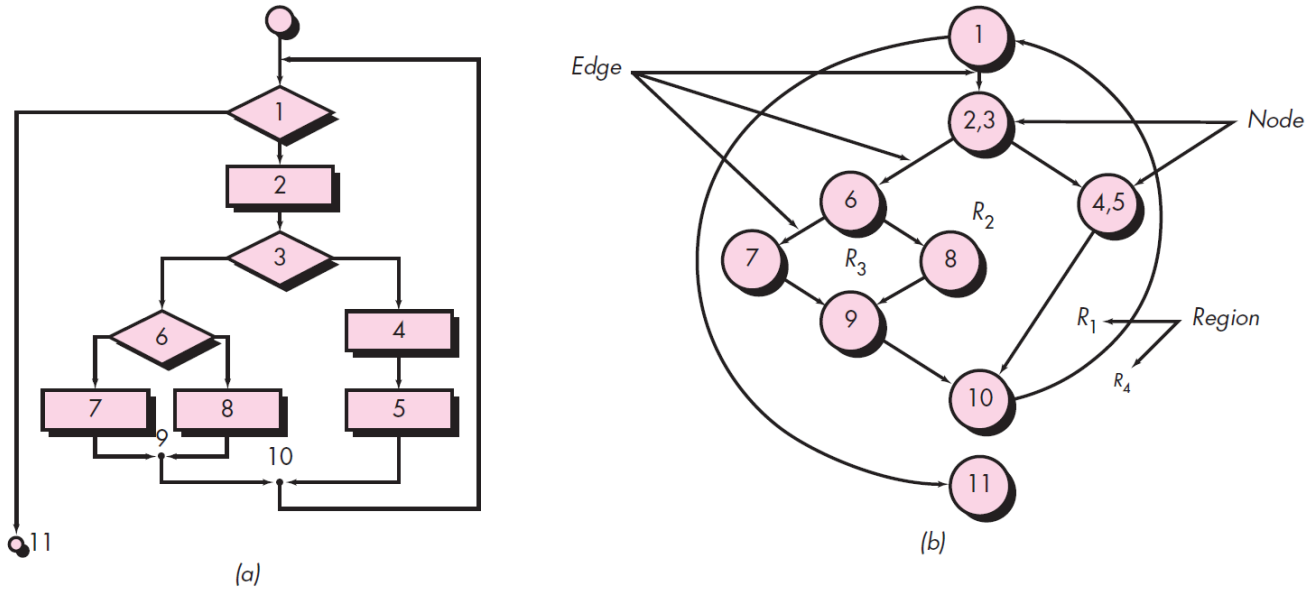
# 23.3 White-Box Testing

23.4 Basis Path Testing

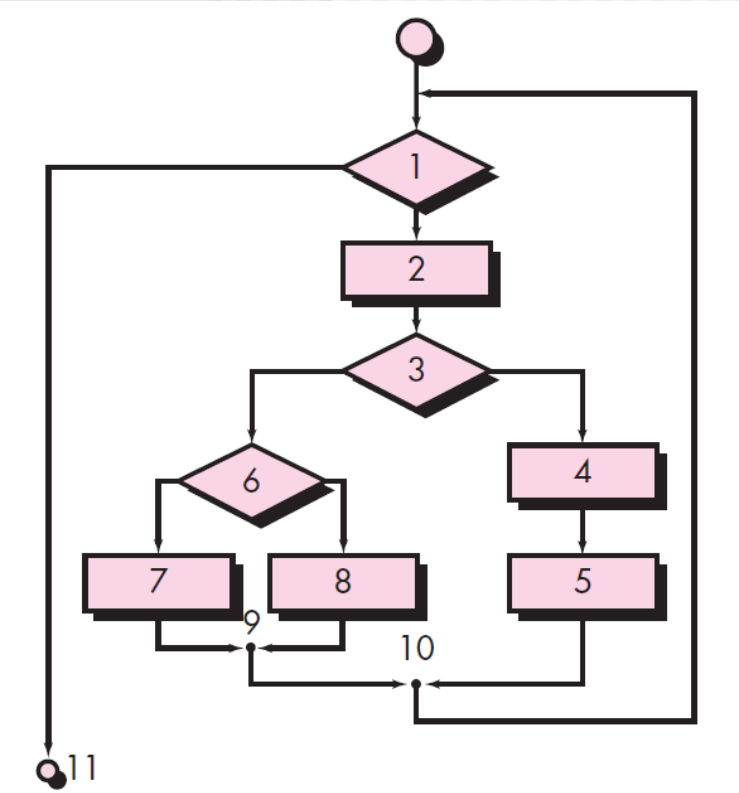
23.5 Control Structure Testing



(a) Flowchart and (b) flow graph



# 23.4 Basis Path Testing




(1) We compute the cyclomatic complexity V:

number of simple decisions + 1

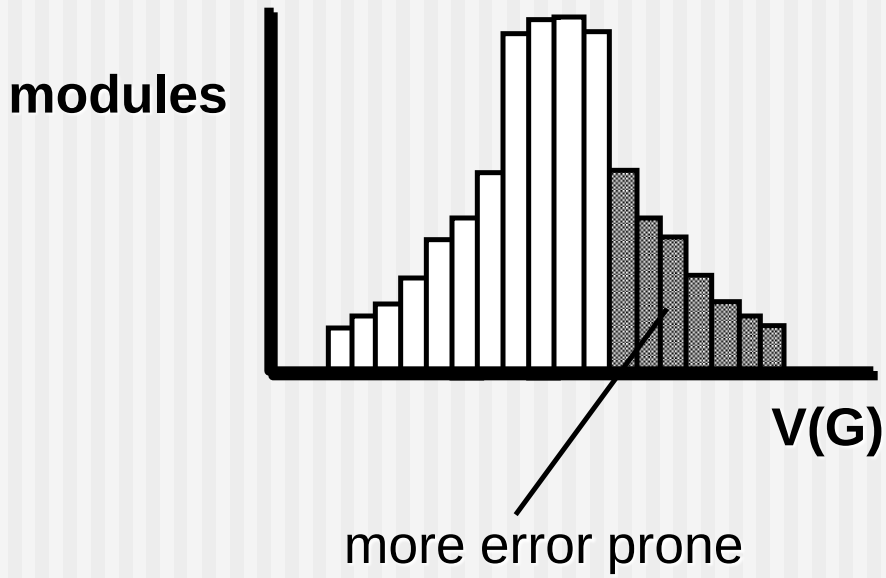
or

number of enclosed areas + 1
of flowchart

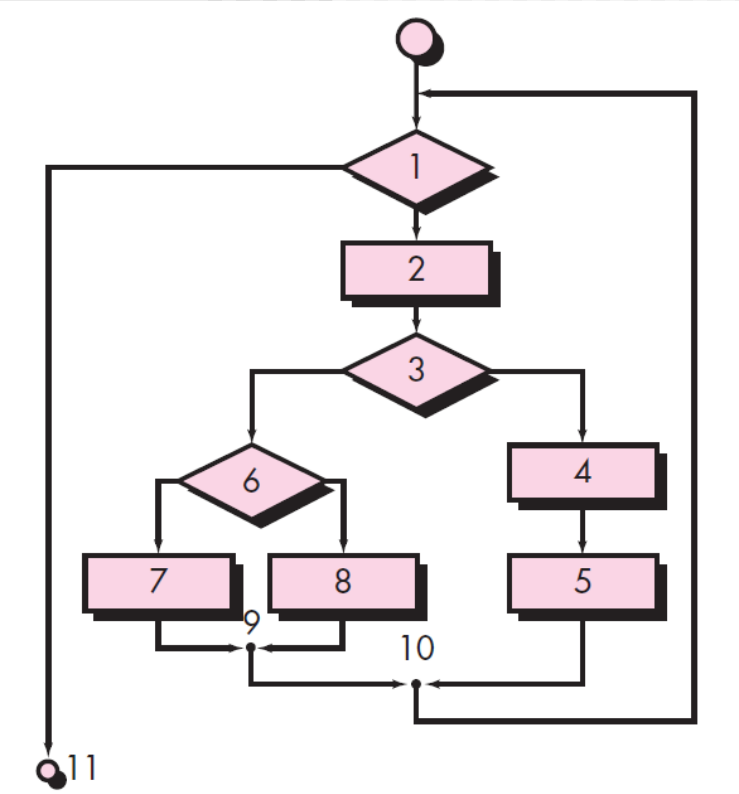
In this case, V(G) = 4

# Cyclomatic Complexity

The higher V(G) is, the higher the probability of errors is.



# Basis Path Testing




(2) We derive the **independent paths**:

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

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

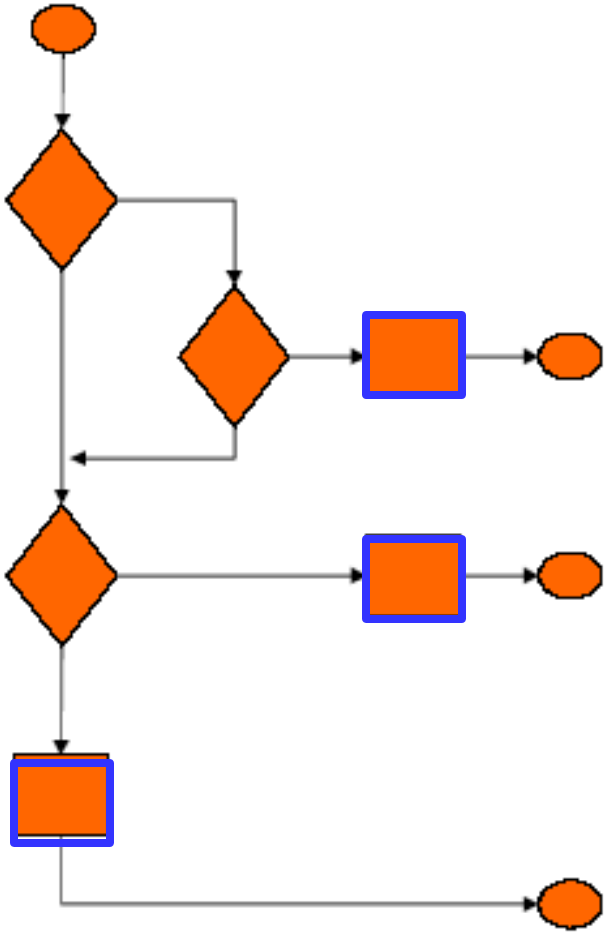
**1,2,3, 4, 5, 10, 1, 2 3, 6, 8, 9, 10, 1, 11**

Is NOT an independent path because it is a combination of already specified paths and does not traverse **any new edges**.

(3) Finally, we derive test cases to exercise these paths.

# White Box: Statement Coverage

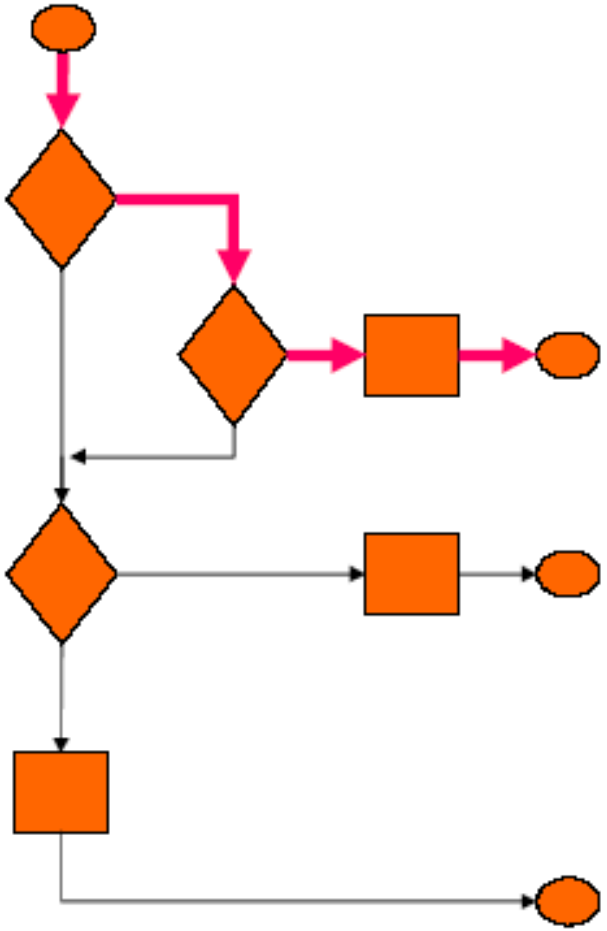
- **Statement coverage**
  - What portion of program statements (nodes) are touched by test cases
- Advantages
  - Test suite size linear in size of code
  - Coverage easily assessed
- Issues
  - Dead code is not reached
  - May require some sophistication to select input sets (McCabe basis paths)
  - Fault-tolerant error-handling code may be difficult to "touch"
  - Metric: Could create incentive to *remove* error handlers!



# White Box: Path Coverage (1/3)

- **Path coverage**
  - What portion of all possible paths through the program are covered by tests?

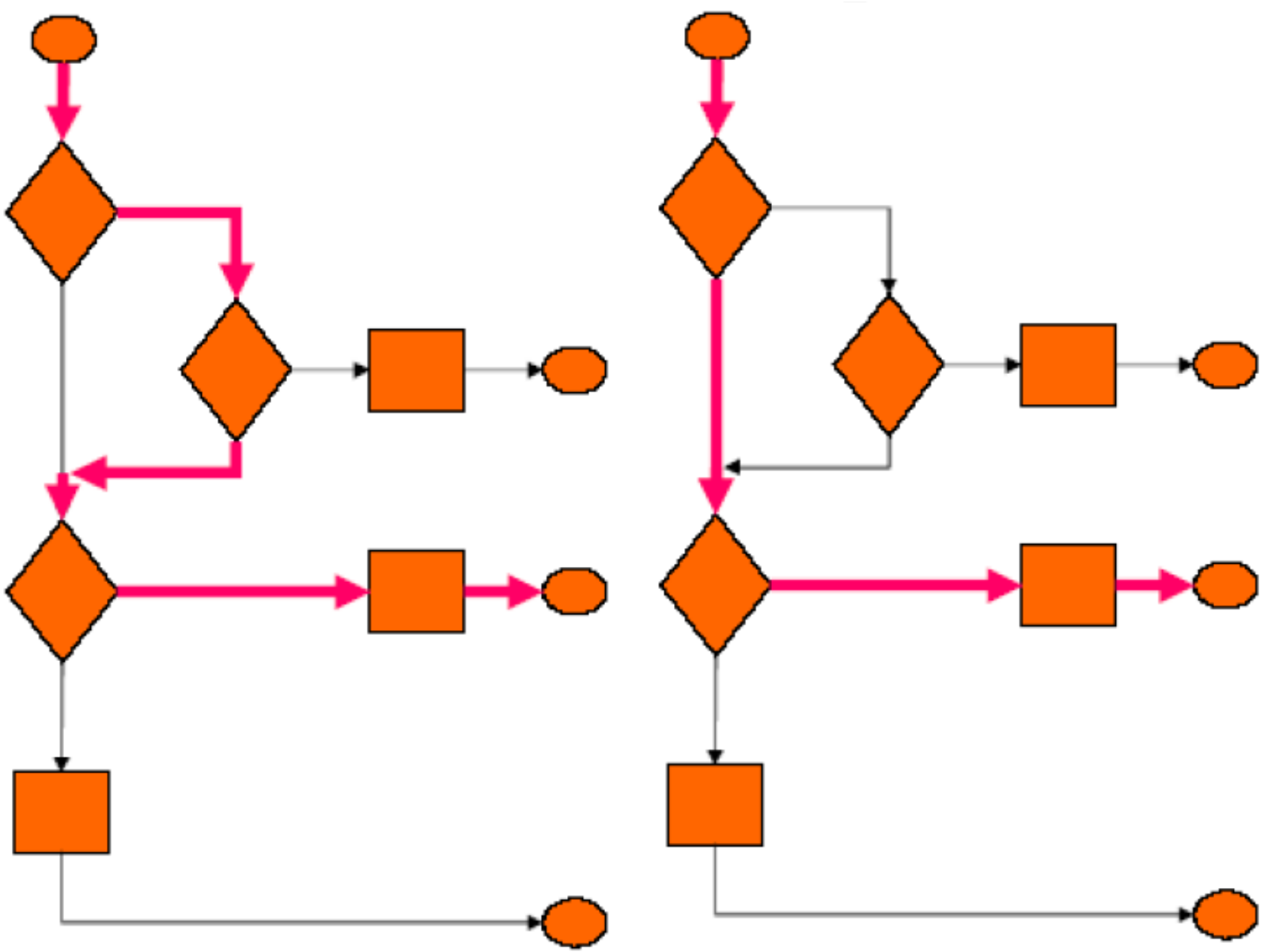
For the flowchart on the right hand side, how many paths are need to satisfy path coverage?



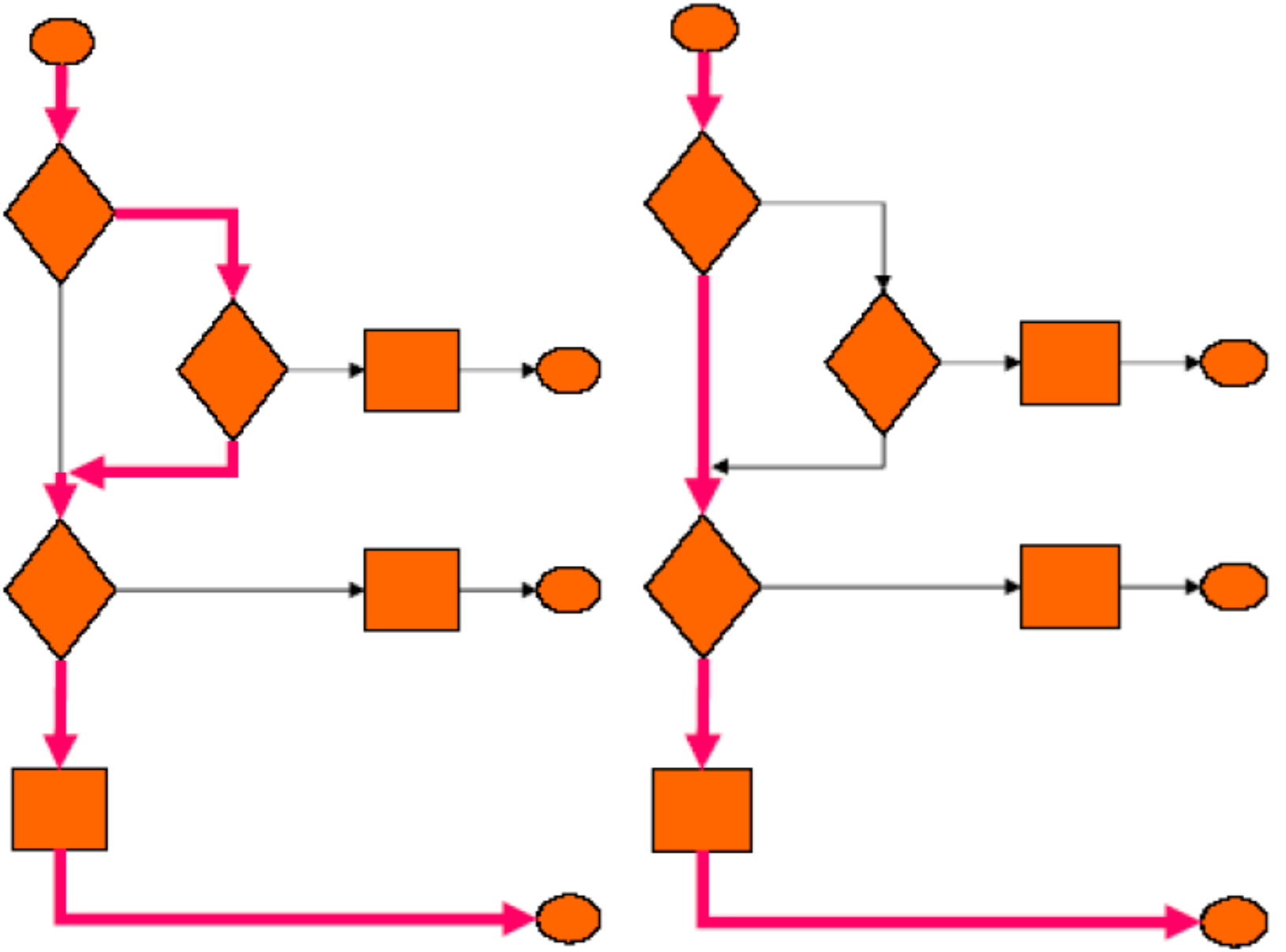
- Advantages
  - Better coverage of logical flows

- Disadvantages
  - Not all paths are possible, or necessary
    - What are the *significant* paths?
  - Combinatorial explosion in cases unless careful choices are made
    - E.g., sequence of *n* **if** tests can yield up to 2^n possible paths
  - Assumption that program structure is basically sound

# White Box: Path Coverage (2/3)



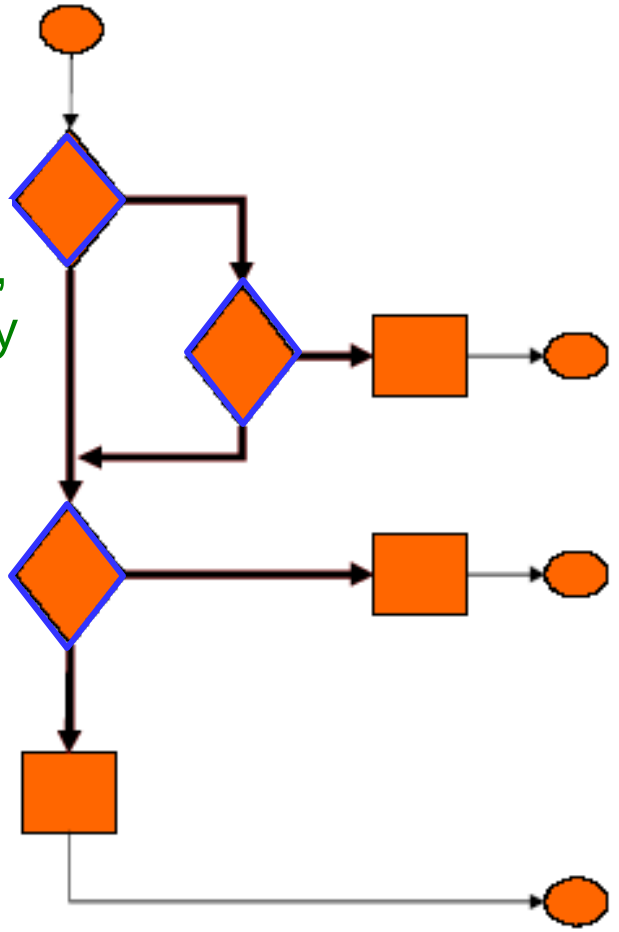
# White Box: Path Coverage (3/3)



# White Box: Branch Coverage

- **Branch coverage**
  - What portion of condition branches are covered by test cases?

For the flowchart on the right hand side, how many test cases are need to satisfy branch coverage?



- Advantages
  - Test suite size and content derived from structure of boolean expressions
  - Coverage easily assessed
- Issues
  - Dead code is not reached
  - Fault-tolerant error-handling code may be difficult to "touch"

# Condition Test (1/3)

- Branch coverage considers only decision results.
- However, in many cases, decisions are made based on multiple conditions.

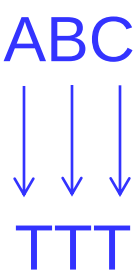
  **Example** 1 IF A < 10 and B > 250 THEN . . .

  **Example** 2 IF (A < 10 or B > 250) THEN . . .

  If the first condition is true, then the truth or falsity of "B > 250" does not matter.

☛ For a thorough testing, it is desirable to test at least once truth or falsity of each condition (Condition Test).

**Example** For "IF (A or B) and C",
{TTT, FFF} is adequate.



# Condition Test (2/3)

- But testing based only on conditions is not thorough.

**Example** To include all of A < 10, A >= 10, B > 250, B <= 250,
we may use (A < 10 and B > 250) and
(A >= 10 and B <= 250) as test cases.
But it only tests "False" decision.

**Example** 1 IF A < 10 and B > 250 THEN . . .
**Example** 2 IF (A < 10 or B > 250) THEN . . .

☛ For a thorough testing, it is desirable to test at least once truth or falsity of each condition and each decision (Condition Decision Test).

**Example** For "IF (A or B) and C",
{TTT, FFF} is adequate.

# Condition Test (3/3)

☛ By requiring testing all combinations of the true case and the false case of each condition, a more thorough testing can be achieved (Multiple Condition Test). => The following 8 test cases are needed.

| | **ABC** | **ABC** | **ABC** | **ABC** | **ABC** | **ABC** | **ABC** | **ABC** |
|---|---|---|---|---|---|---|---|---|
| Truth Vectors | TTT | TTF | TFT | TFF | FTT | FTF | FFT | FFF |
| Decision Outcome | T | F | T | F | T | F | F | F |

- There are many test cases.

☛ Delete test cases so that both true and false cases of each condition and each decision are tested (Modified Condition Decision Test).
=> 6 test cases in the white background will be sufficient.
(Even fewer than 6 test cases will do.)

# 23.5 Control Structure Testing

- Control Flow Testing
  - Path testing
  - Branch testing
  - Condition testing
  - Loop testing ☚ To be discussed in this section
- Data flow testing
  - selects test paths according to the locations of definitions and uses of variables

# Data Flow Testing [Fra93]

- Selects test paths of a program according to the locations of definitions and uses of variables.

  - For a statement *S*,
    - DEF(*S*) = {*X* | *S* contains a definition of *X*}
    - USE(*S*) = {*X* | *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 statements,
    *X* is in DEF(*S*) and USE(*S′*), and
    the definition of *X* in *S* is live at *S′*

- Derive a test case for each [*X, S, S′*].

# Data Flow Testing – Example

```
totalLocks = 0
totalStocks = 0
totalBarrels = 0
Input(locks)
while NOT(locks = -1) //loop condition uses -1
        Input(stocks, barrels)
        totalLocks = totalLocks + locks
        totalStocks = totalStocks + stocks
        totalBarrels = totalBarrels + barrels
        Input(locks)
endwhile
Output("Locks sold: ", totalLocks)
```

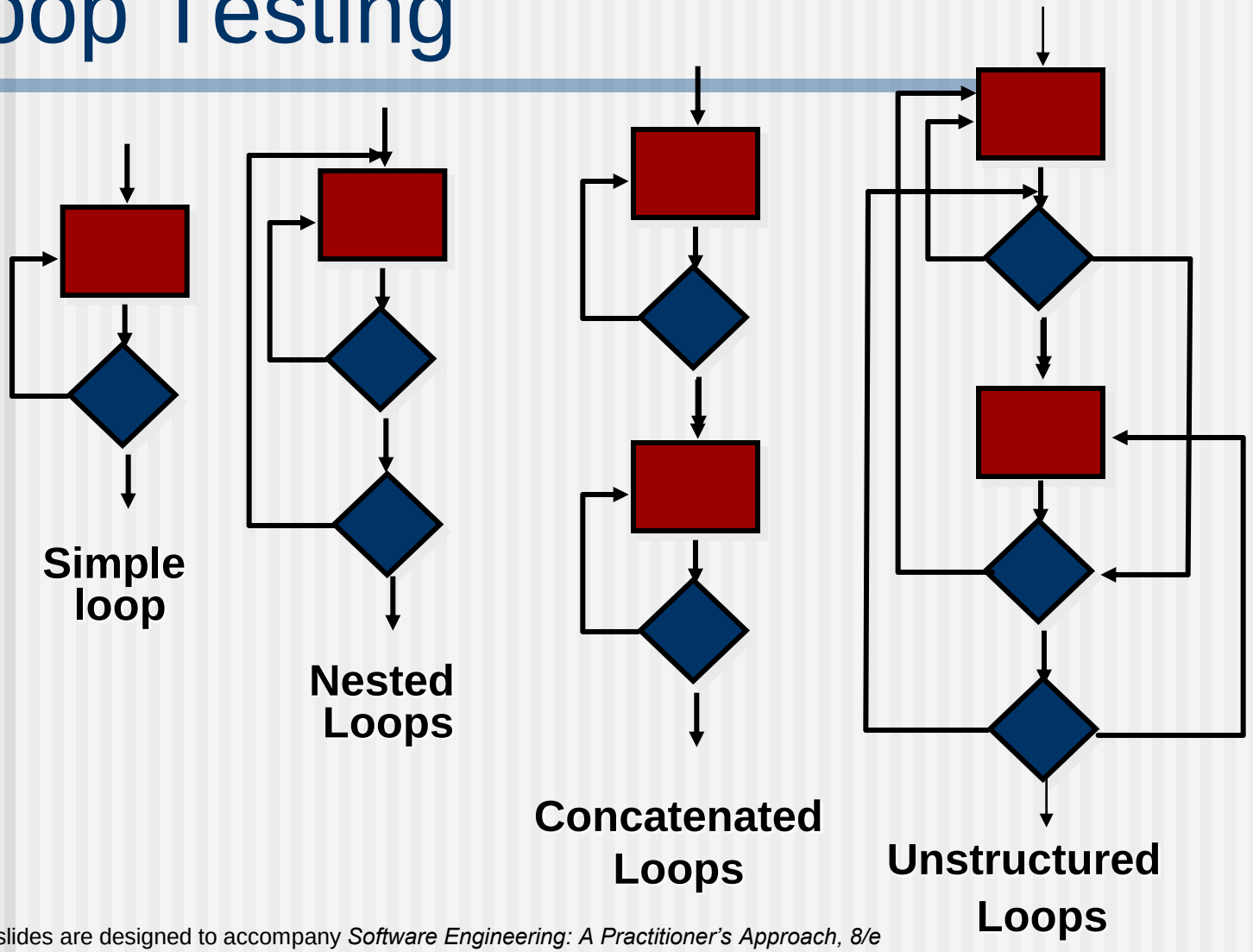
# Data Flow Testing - Predicate/Computation Use

```
6. t1 = b + c
7. t2 = a + c
8. t3 = a + b
9. if (a < t1) and (b < t2) and < t3)
10.     then IsATriangle = True
11.     else IsATriangle = False
12. endif

...
```

- USE(v, n) can be classified as
  - Predicate use (P-use)
  - Computation use(C-use)
- USE(a, 7) ?
- USE(a, 9)?

# Loop Testing



# Loop Testing: Simple Loops

**Minimum conditions**

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

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.

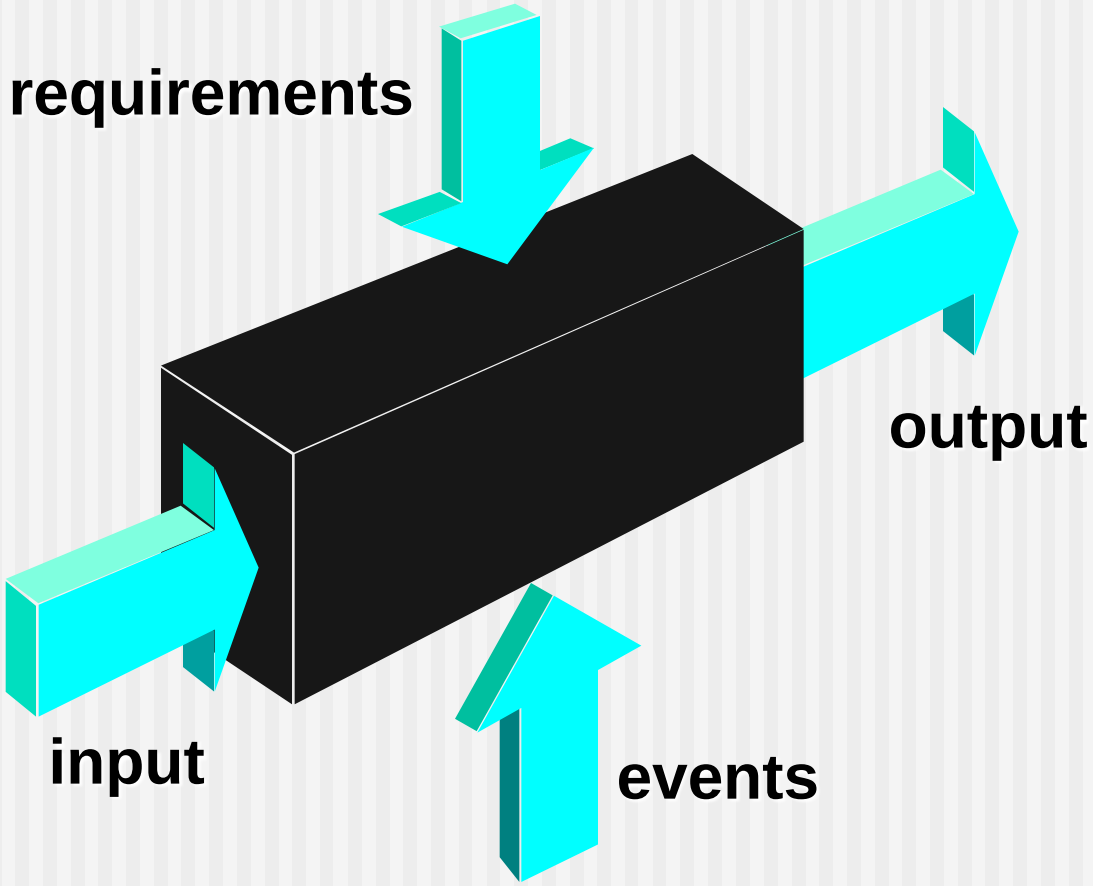
# Loop Testing: Concatenated Loops

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

**Example**

The final loop counter value of loop 1 is used to initialize loop 2.

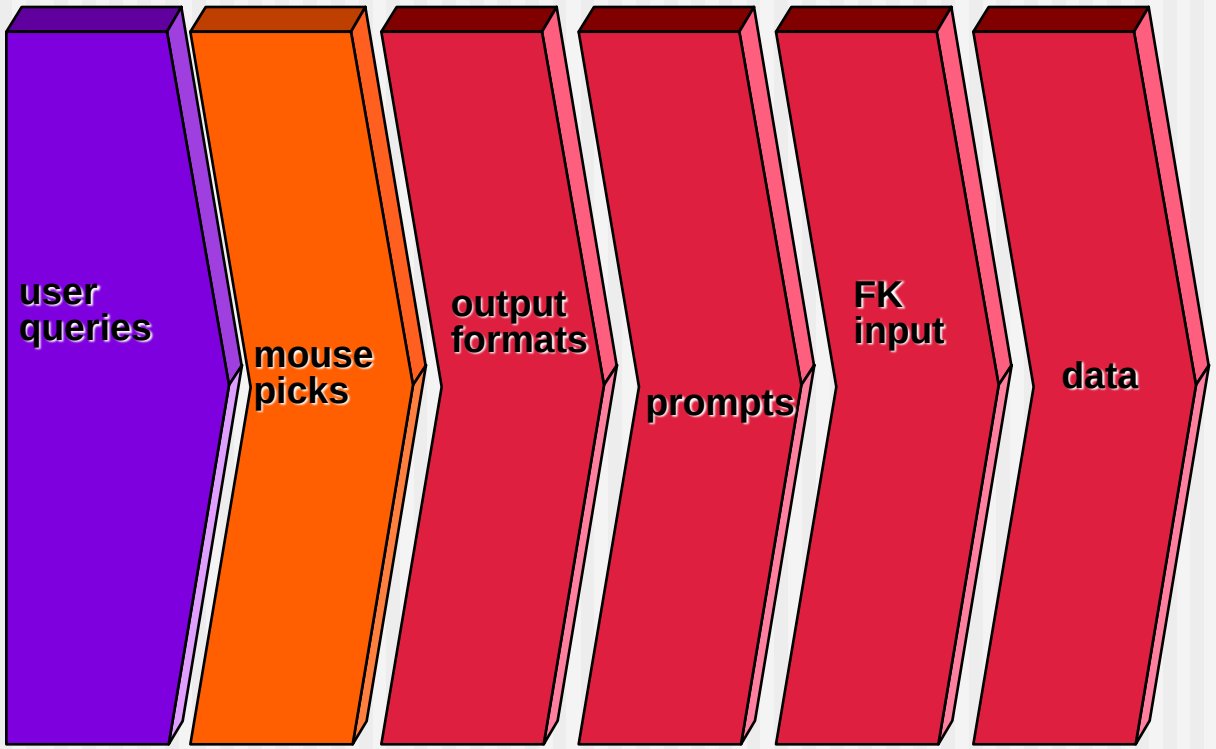
# 23.6 Black-Box Testing



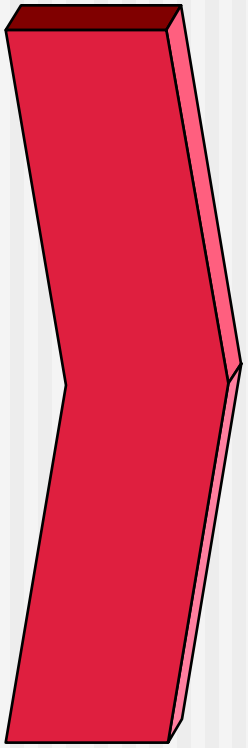
# Black-Box Testing

- 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?

# 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)

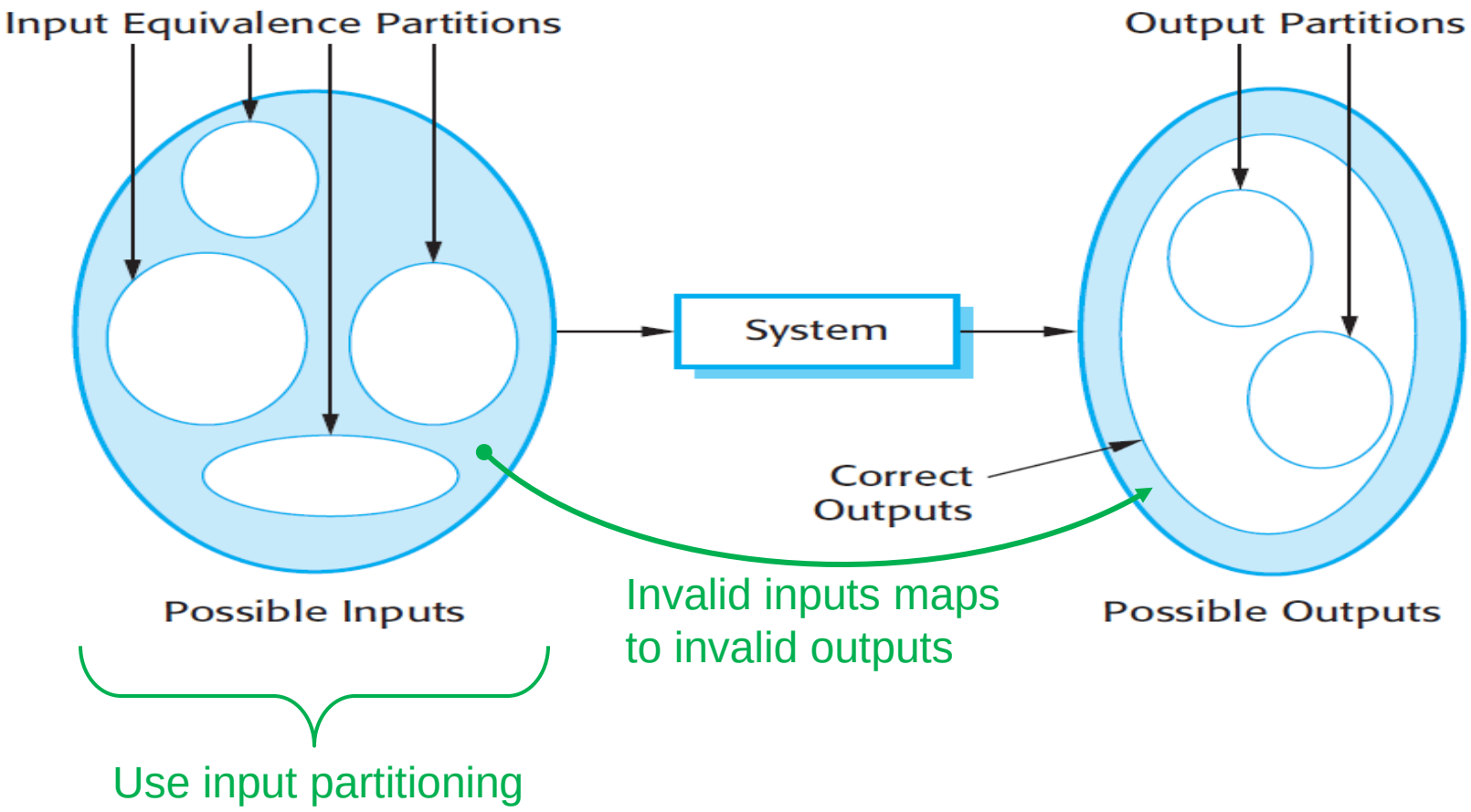
***Invalid data***

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

# Equivalence Partitioning (1/6)

- Input data and output results often fall into different classes where all members of a class are related
- Each of these classes is an equivalence partition where the program behaves in an equivalent way for each class member
- Test cases should be chosen from each partition

# Equivalence Partitioning (2/6)

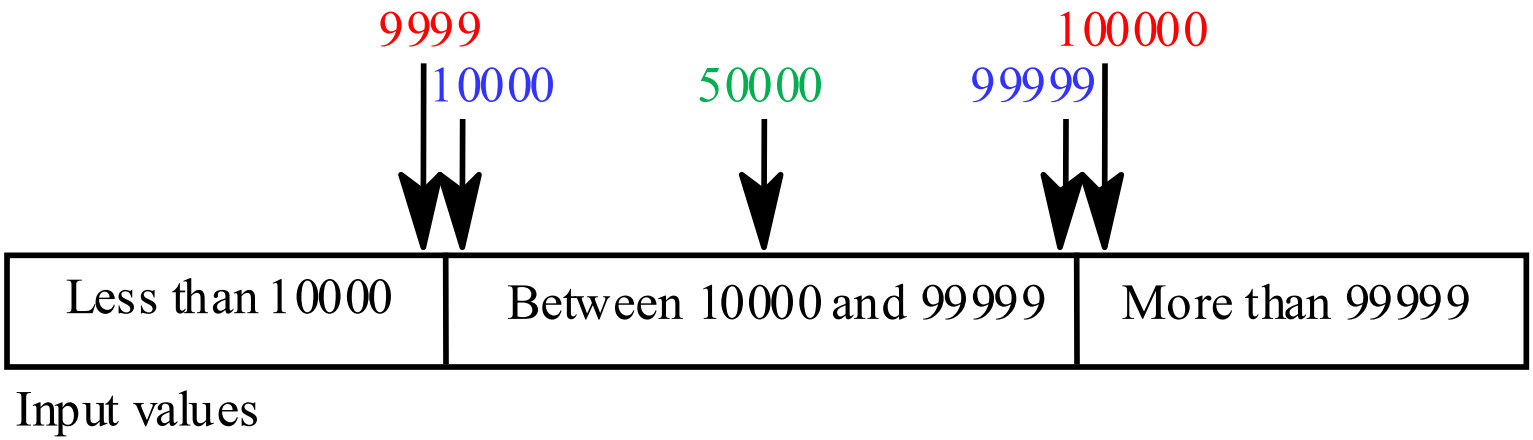


# Equivalence Partitioning (3/6)

## Example 1.

- Partition system inputs and outputs into ‘equivalence sets’
  - If input is a 5-digit integer between 10,000 and 99,999, equivalence partitions are:

    < 10000, 10000, 10000< & < 99999, 99999, 99999 <
- Choose test cases at the boundary of these sets



# Equivalence Partitioning (4/6)

## Example 2. Search Routine

How many inputs are in this program?

Specification

```
procedure Search (Key : ELEM ; T: ELEM_ARRAY;
        Found : in out BOOLEAN; L: in out ELEM_INDEX) ;

Pre-condition
        -- the array has at least one element
        FIRST <= LAST
Post-condition
        -- the element is found and is referenced by L
        ( Found and T(L) = Key)
or
        -- the element is not in the array
        (not Found and
        not (exists i, FIRST<=i<=LAST, T(i) = Key ))
```

# Equivalence Partitioning (5/6)

- Input Partitions
  - Inputs which conform to the pre-conditions
  - Inputs where a pre-condition does not hold
    => Tests robustness, not functional correctness
  - Inputs where the key element is a member of the array
  - Inputs where the key element is not a member of the array

- Guidelines for testing sequences (= Test Requirements)
  - Test software with sequences which have only a single value
  - Use sequences of different sizes in different tests
  - Derive tests so that the first, middle and last elements of the sequence are accessed

# Equivalence Partitioning (6/6)

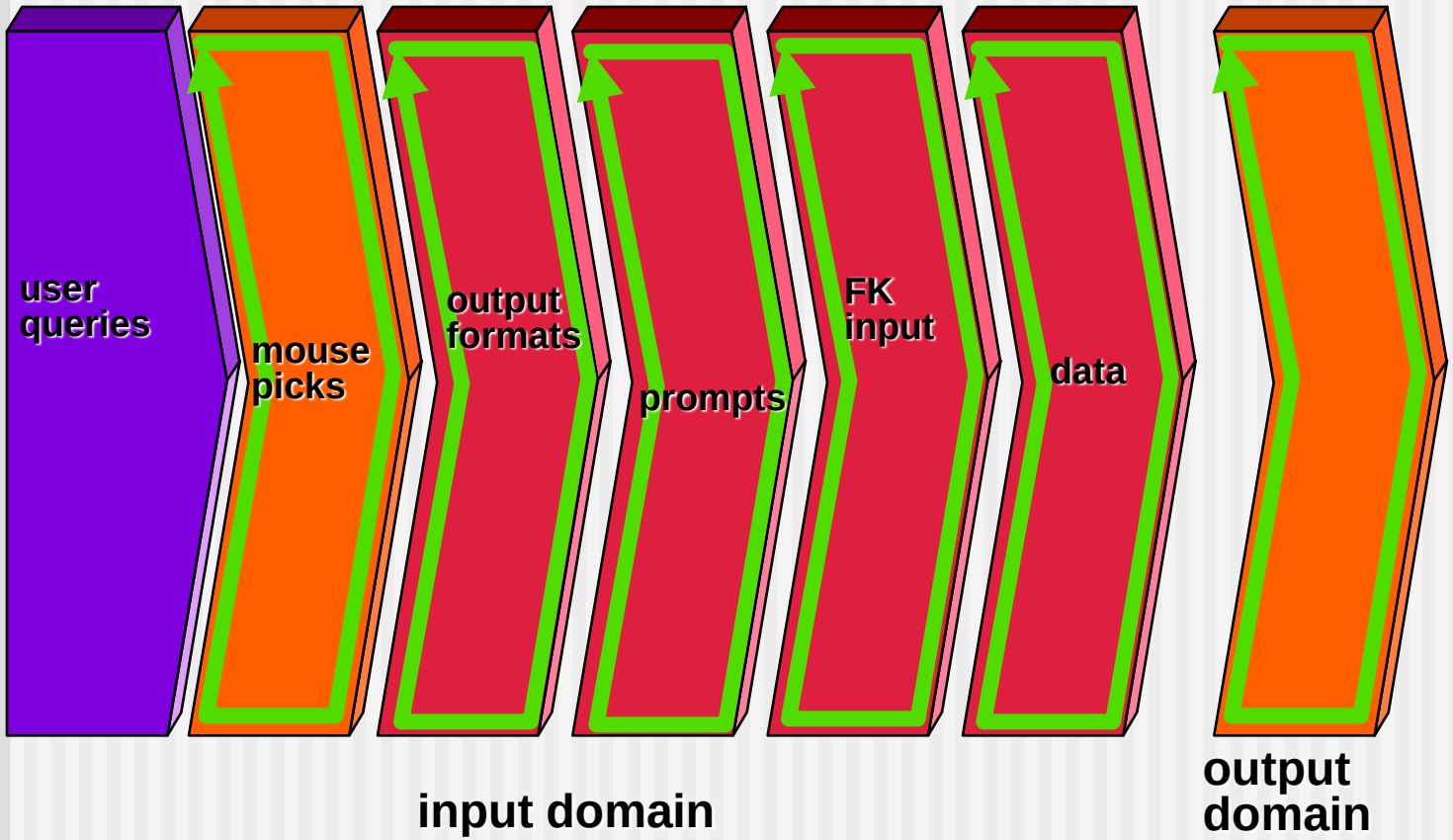
## Test Data Design

| Array | Element |
|---|---|
| Single value | In sequence |
| Single value | Not in sequence |
| More than 1 value | First element in sequence |
| More than 1 value | Last element in sequence |
| More than 1 value | Middle element in sequence |
| More than 1 value | Not in sequence |

## Test Data Determination

| **Input sequence** (T) | **Key** (Key) | **Output** (Found, L) |
|---|---|---|
| 17 | 17 | true, 1 |
| 17 | 0 | false, ?? |
| **17** 29, 21, 23 | 17 | true, 1 |
| 41, 18, 9, 31, 30, 16, **45** | 45 | true, 7 |
| 17, 18, 21, **23**, 29, 41, 38 | 23 | true, 4 |
| 21, 23, 29, 33, 38 | 25 | false, ?? |

# Boundary Value Analysis



# Comparison Testing

- Used only when the reliability of software is absolutely critical

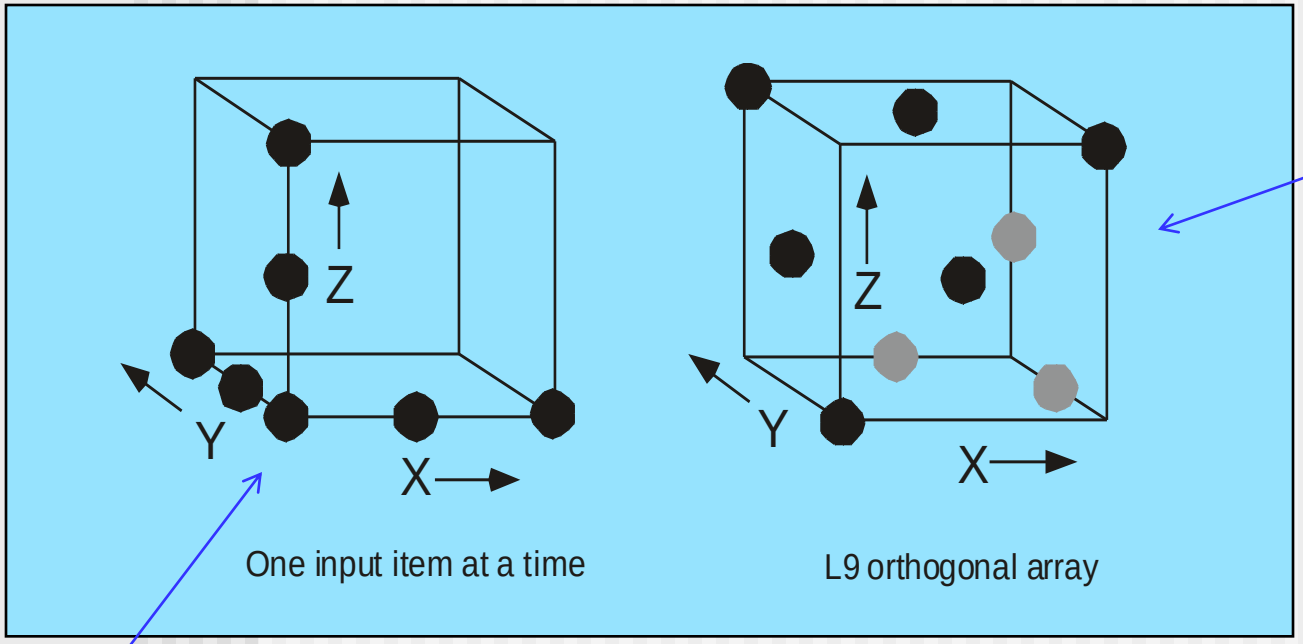
  - Separate software engineering teams develop independent versions of an application using the same specification

  - 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

# Orthogonal Array Testing

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




Dots are dispersed uniformly throughout the test domain

☛ Covers some of multimode faults.

Uncovers only single mode faults, i.e. faults triggered by a single parameter?

If we cannot afford to use 81 test cases, which one is better?

$L9(3^4)$

| Test case | Test parameters | | | |
|---|---|---|---|---|
| | P1 | P2 | P3 | P4 |
| 1 | **1** | **1** | **1** | **1** |
| 2 | **2** | **1** | **1** | **1** |
| 3 | **3** | **1** | **1** | **1** |
| 4 | **1** | **2** | **1** | **1** |
| 5 | **1** | **3** | **1** | **1** |
| 6 | **1** | **1** | **2** | **1** |
| 7 | **1** | **1** | **3** | **1** |
| 8 | **1** | **1** | **1** | **2** |
| 9 | **1** | **1** | **1** | **3** |

| Test case | Test parameters | | | |
|---|---|---|---|---|
| | P1 | P2 | P3 | P4 |
| 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 2 | 2 | 2 |
| 3 | 1 | 3 | 3 | 3 |
| 4 | 2 | 1 | 2 | 3 |
| 5 | 2 | 2 | 3 | 1 |
| 6 | 2 | 3 | 1 | 2 |
| 7 | 3 | 1 | 3 | 2 |
| 8 | 3 | 2 | 1 | 3 |
| 9 | 3 | 3 | 2 | 1 |

Uncovers only single mode faults, i.e. faults triggered by a single parameter?

# 6. Decision Table Based Testing

## Decision Table Based Testing

- Decision table guarantees that every possible combination of condition values is to be considered.
- Ideal for describing situations in which a number of combination of actions are taken under sets of conditions.
- When conditions are taken as inputs, and actions as outputs, the rules can be interpreted as test cases.

# 6. Decision Table Based Testing

## Decision Table for the Example

| C1: a < b + c? | F | T | T | T | T | T | T | T | T | T | T |
|---|---|---|---|---|---|---|---|---|---|---|---|
| C2: b < a + c? | - | F | T | T | T | T | T | T | T | T | T |
| C3: c < a + b? | - | - | F | T | T | T | T | T | T | T | T |
| C4: a = b? | - | - | - | T | T | T | T | F | F | F | F |
| C5: a = c? | - | - | - | T | T | F | F | T | T | F | F |
| C6: b = c? | - | - | - | T | F | T | F | T | F | T | F |
| A1: not a triangle | X | X | X | | | | | | | | |
| A2: Scalene | | | | | | | | | | | X |
| A3: Isosceles | | | | | | | X | | X | X | |
| A4: Equilateral | | | | X | | | | | | | |
| A5: Impossible | | | | | X | X | | X | | | |

# 6. Decision Table Based Testing

## Test Cases for the Example

| Test Cases | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| C1: a < b + c? | F | T | T | T | T | T | T | T | T | T | T |
| C2: b < a + c? | - | F | T | T | T | T | T | T | T | T | T |
| C3: c < a + b? | - | - | F | T | T | T | T | T | T | T | T |
| C4: a = b? | - | - | - | T | T | T | T | F | F | F | F |
| C5: a = c? | - | - | - | T | T | F | F | T | T | F | F |
| C6: b = c? | - | - | - | T | F | T | F | T | F | T | F |
| a | 4 | 1 | 1 | 5 | ? | ? | 2 | ? | 2 | 3 | 3 |
| b | 1 | 4 | 2 | 5 | ? | ? | 2 | ? | 3 | 2 | 4 |
| c | 2 | 2 | 4 | 5 | ? | ? | 3 | ? | 2 | 2 | 5 |
| Expected Output | No | No | No | Eq | Im | Im | Iso | Im | Iso | Iso | Sc |

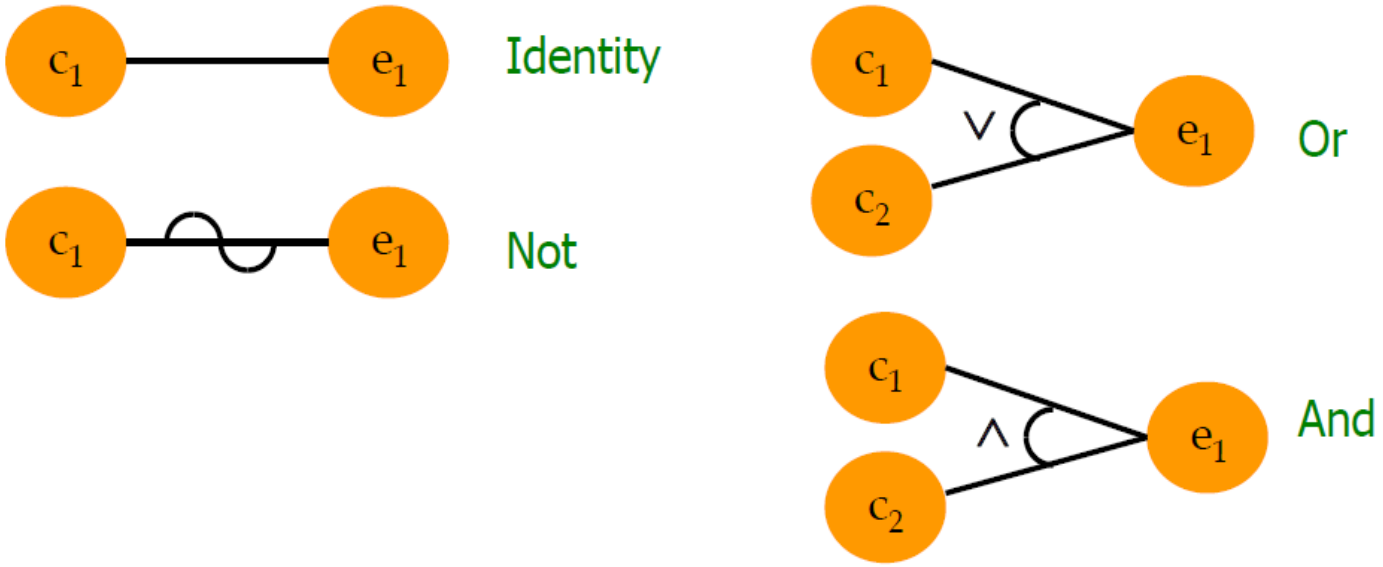
# 6. Decision Table Based Testing

## Cause Effect Graphing

- A techniques for transforming natural language specifications into a more structured and formal specification.
- Provides a concise representation of logical conditions (causes) and corresponding actions (effects).

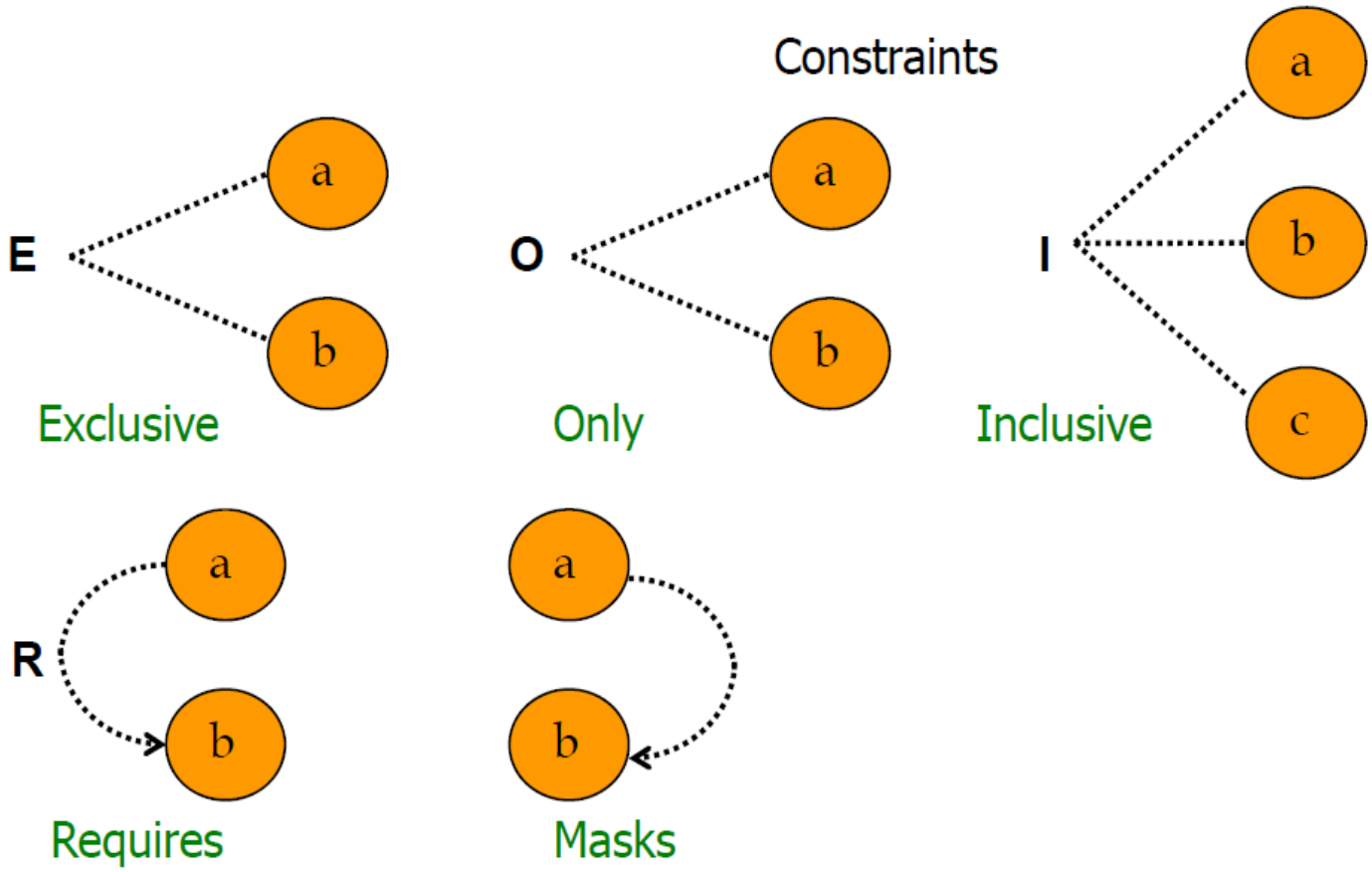
# 6. Decision Table Based Testing

## Notations for Cause Effect Graph



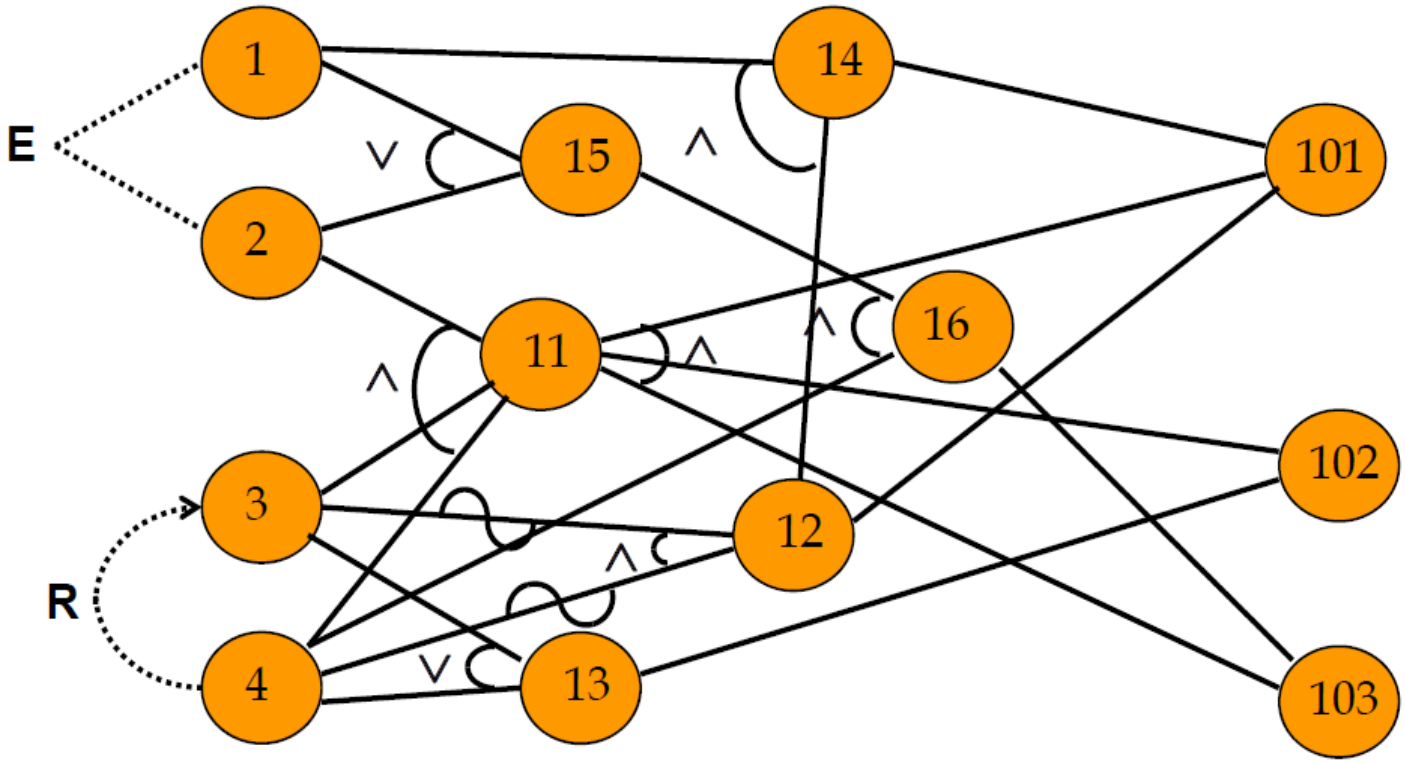
# 6. Decision Table Based Testing

## Notations for Cause Effect Graph (cont'd)



# 6. Decision Table Based Testing

## an Example of Cause-Effect Graph



# 6. Decision Table Based Testing

## Decision Table for the Example

Causes

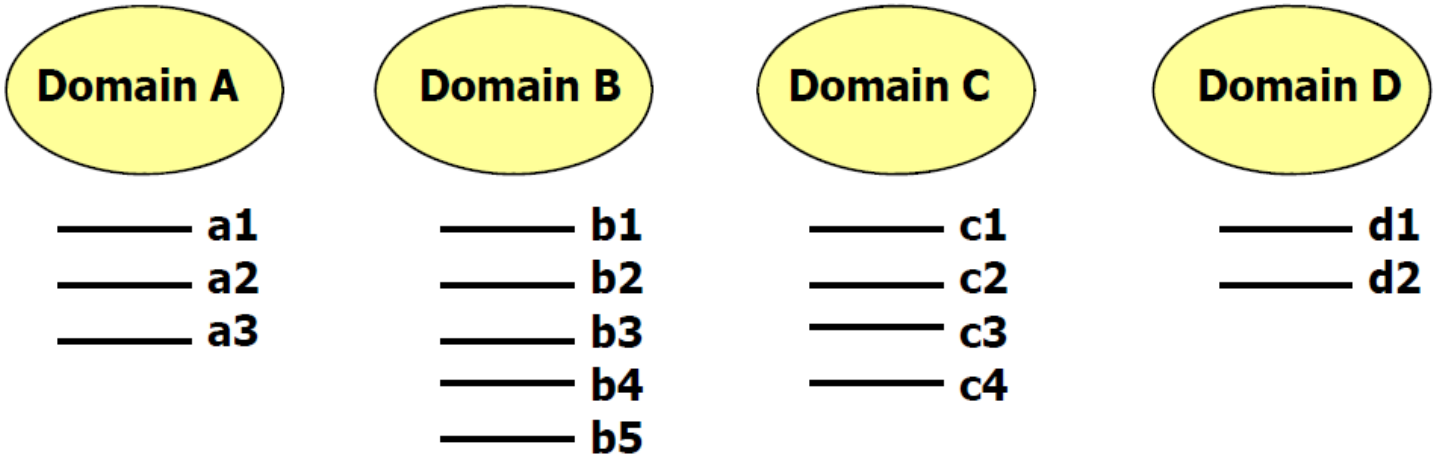
Intermediate Causes

Effects

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| 1 | 1 | | | | | | | 0 | 1 |
| 2 | | 1 | | 1 | | | 1 | 1 | 0 |
| 3 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | | |
| 4 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 | 1 |
| | | | | | | | | | |
| 11 | | 1 | | 1 | | | 1 | | |
| 12 | 1 | | 1 | | | | | | |
| 13 | | | | | 1 | 1 | | | |
| 14 | 1 | | | | | | | | |
| 15 | | | | | | | | 1 | 1 |
| 16 | | | | | | | | 1 | 1 |
| | | | | | | | | | |
| 101 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| 102 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| 103 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 1 |

# 7. Combinatorial Testing

## How to Combine Input Parameter Values



# 7. Combinatorial Testing

## Weak Equivalence Class (Default) Testing

- Select one value from each equivalence class.

| Test Case | a | b | c | d |
|---|---|---|---|---|
| tc1 | a1 | b1 | c1 | d1 |
| tc2 | a2 | b2 | c2 | d2 |
| tc3 | a3 | b3 | c3 | d1 |
| tc4 | a1 | b4 | c4 | d2 |
| tc5 | a2 | b5 | c1 | d1 |

# 7. Combinatorial Testing

## Strong Equivalence Class Testing

- Take the Cartesian product of the partition subsets.
  - 120 test cases for the example
- Is suitable for a case in which physical variables have numerous interactions and failure of the function is extremely costly.
  - ☞ Worst Case Analysis
  - ☞ Common Mode Failure

# 7. Combinatorial Testing

## Pairwise Testing

- Requires that for each pair of input parameters of a system, every combination of valid values of these two parameters be covered by at least one test case.
  - ☞ All Pairs Testing, N-way Testing, Combinatorial Testing, Orthogonal Arrays
- The number of tests required to cover all pairwise parameter combinations grows logarithmically with the number of parameters.
- Can be supported by the automated tools
  - ☞ AETG, Combinatorial Test Service (IBM)

# 7. Combinatorial Testing

## Pairwise Testing

| Test Case | a | b | c | d | Test Case | a | b | c | d |
|---|---|---|---|---|---|---|---|---|---|
| tc1 | a1 | b1 | c1 | d1 | tc11 | a3 | b1 | c3 | ~d1 |
| tc2 | a2 | b1 | c2 | d2 | tc12 | a3 | b2 | c4 | ~d2 |
| tc3 | a2 | b2 | c1 | d1 | tc13 | a1 | b3 | c3 | ~d2 |
| tc4 | a1 | b2 | c2 | d2 | tc14 | a2 | b3 | c4 | ~d1 |
| tc5 | a3 | b3 | c1 | d2 | tc15 | a3 | b4 | c1 | ~d2 |
| tc6 | a3 | b3 | c2 | d1 | tc16 | a3 | b5 | c2 | ~d1 |
| tc7 | a1 | b4 | c3 | d1 | tc17 | ~a1 | b1 | c4 | ~d2 |
| tc8 | a2 | b4 | c4 | d2 | tc18 | ~a2 | b2 | c3 | ~d1 |
| tc9 | a2 | b5 | c3 | d2 | tc19 | ~a1 | b4 | c2 | ~d1 |
| tc10 | a1 | b5 | c4 | d1 | tc20 | ~a1 | b5 | c1 | ~d2 |

# 7. Combinatorial Testing

## An example of an orthogonal arrays of strength 2

- Strength is the number of columns where we are guaranteed to see all the possibilities an equal number of times.
- 2-level arrays since only 0's and 1's appear.
- 12 rows mean 12 different samples.

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 1 |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |

# 7. Combinatorial Testing

## Issues for Pairwise Test Cases

- The number of tests required to cover all n-way parameter combinations, for fixed n, grows logarithmically in the number of parameters.
- The key concept of how program input variables interact to create outputs is missing for the pairwise testing.
- A white box test set is almost certainly guaranteed to be smaller than the pairwise test set.
  - ☞ A single execution of a path through the code.

# 7. Combinatorial Testing

## How to Combine Input Parameter Values

- Default Testing (Weak Analysis)
  - Based on the Single Fault Assumption (Failures are rarely the result of the simultaneous faults.)
- Pairwise or N-Ways
  - Assumes that troublesome faults are caused by the interactions of a few test parameters.
- Worst Case Analysis (Strong Analysis)
  - Is suitable for a case in which physical variables have numerous interactions and failure of the function is extremely costly.
  - ☞ Common Mode Failure

# 23.7 Model-Based Testing

**Cf. Graph-based Testing (Next slide)**

- Analyze an existing behavioral model for the software or create one.
  - *Behavioral model* indicates how software responds 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.
- Note the expected outputs from the behavioral model 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.

# Graph-Based Methods

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