

Chapter 14

■ **Software Testing Techniques**

Slide Set to accompany

Software Engineering: A Practitioner's Approach, 6/e
by Roger S. Pressman

Slides copyright © 1996, 2001, 2005 by Roger S. Pressman

For non-profit educational use only

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach, 6/e*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information **MUST** appear if these slides are posted on a website for student use.

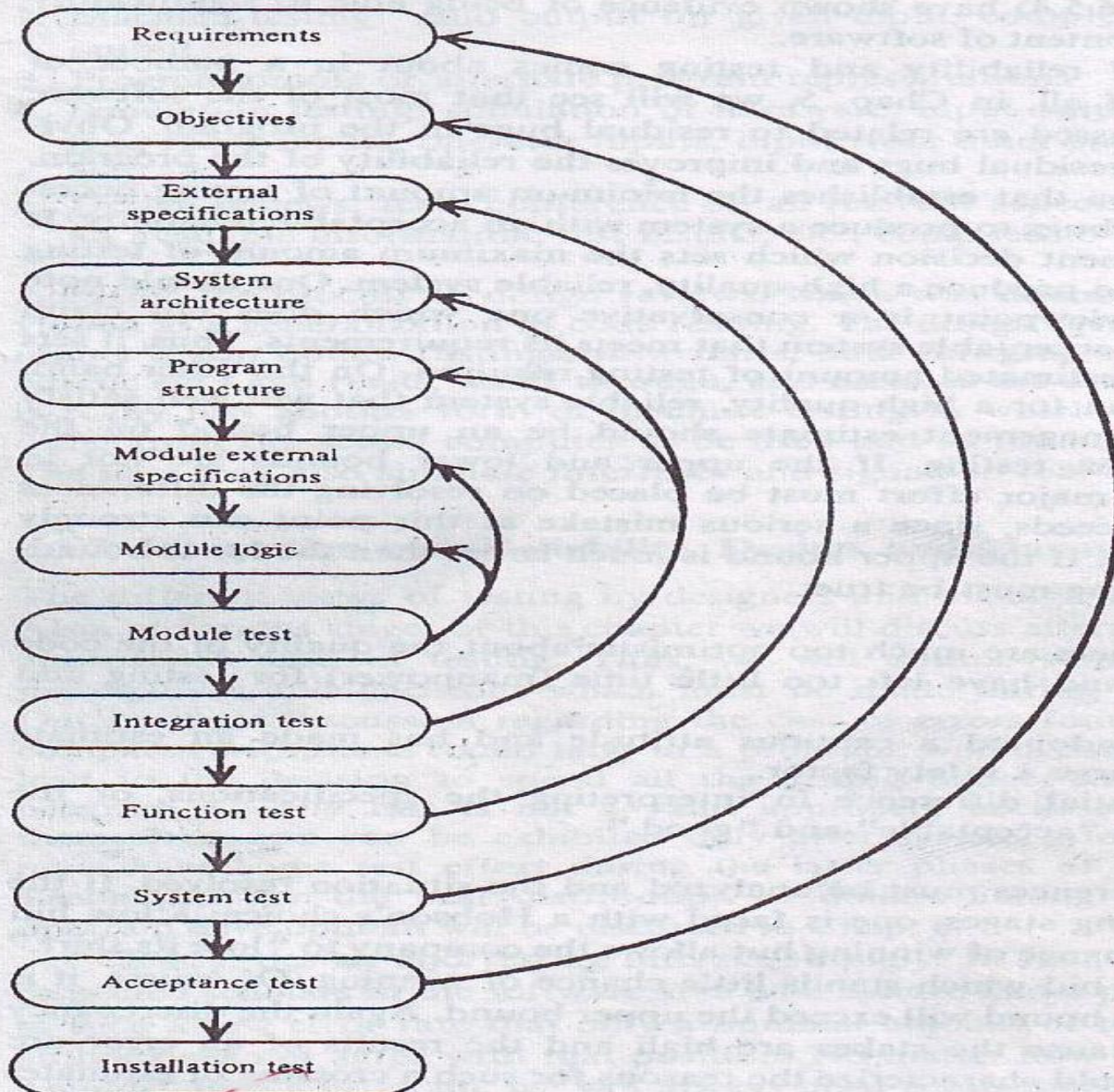
Basic Terminology

- Testing:
- Verification:
- Validation:
- Debugging:

Testing Definitions

- Module/Unit Test:
- Integration Test :
- Function Test :
- System Test :
- Acceptance Test:
- Installation Test:

Relationships of Testing Processes with Design Processes



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

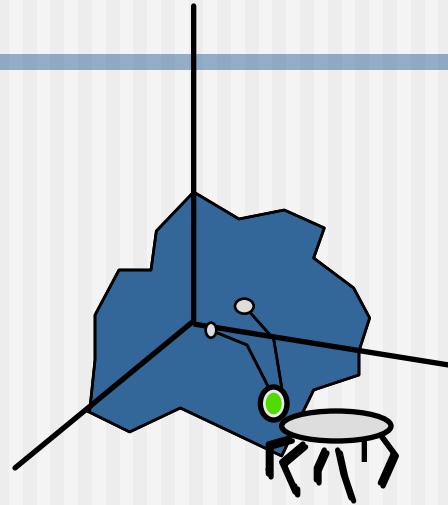
Internal and External Views

- 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

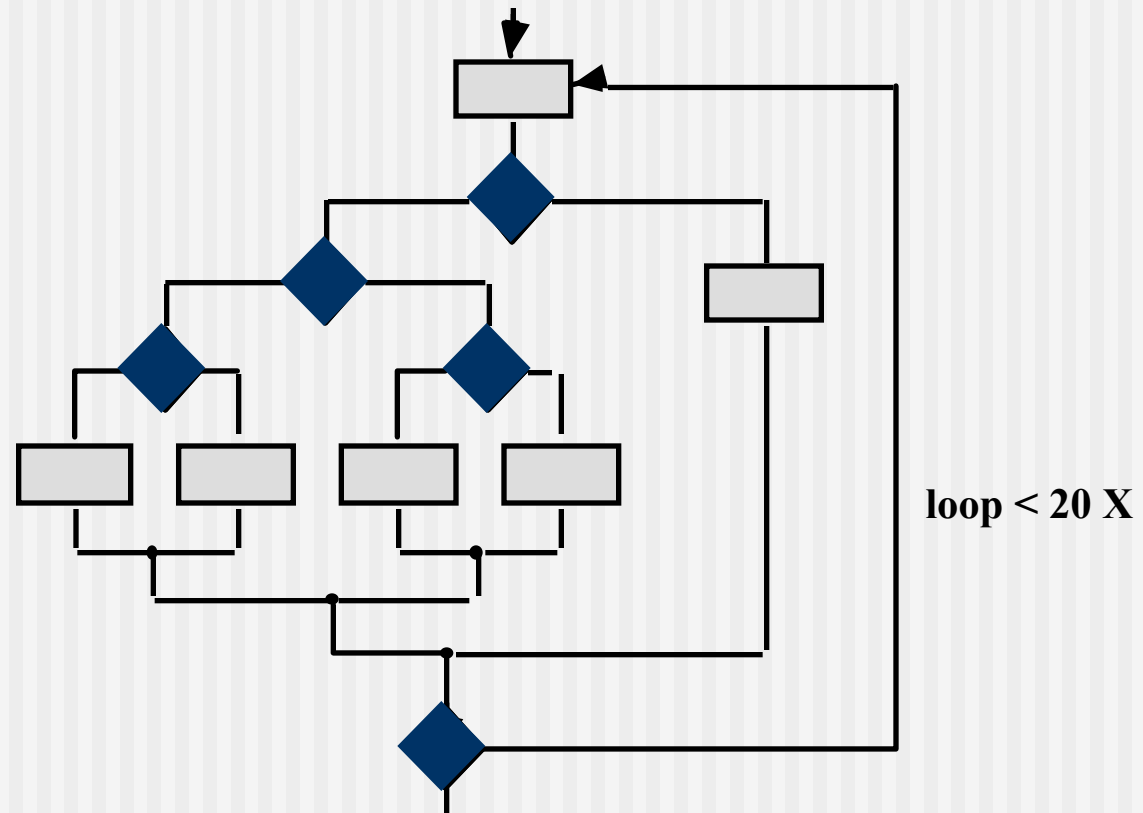


OBJECTIVE to uncover errors

CRITERIA in a complete manner

CONSTRAINT with a minimum of effort and time

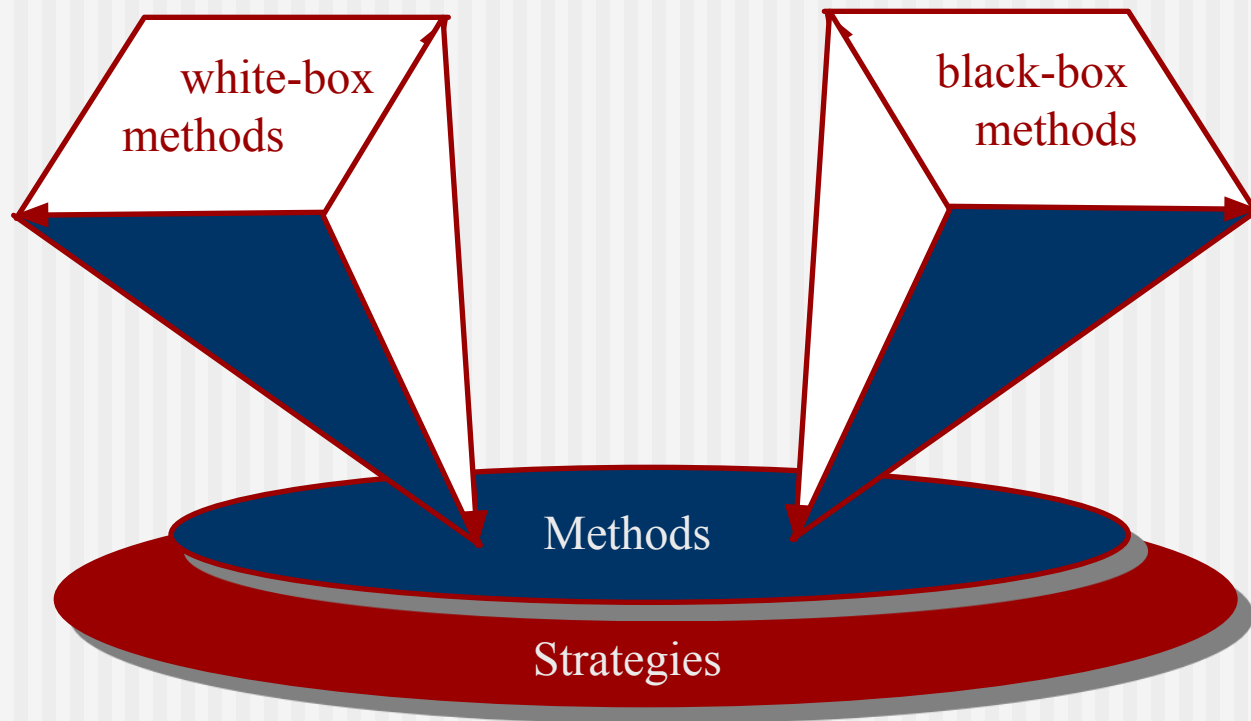
Exhaustive Testing



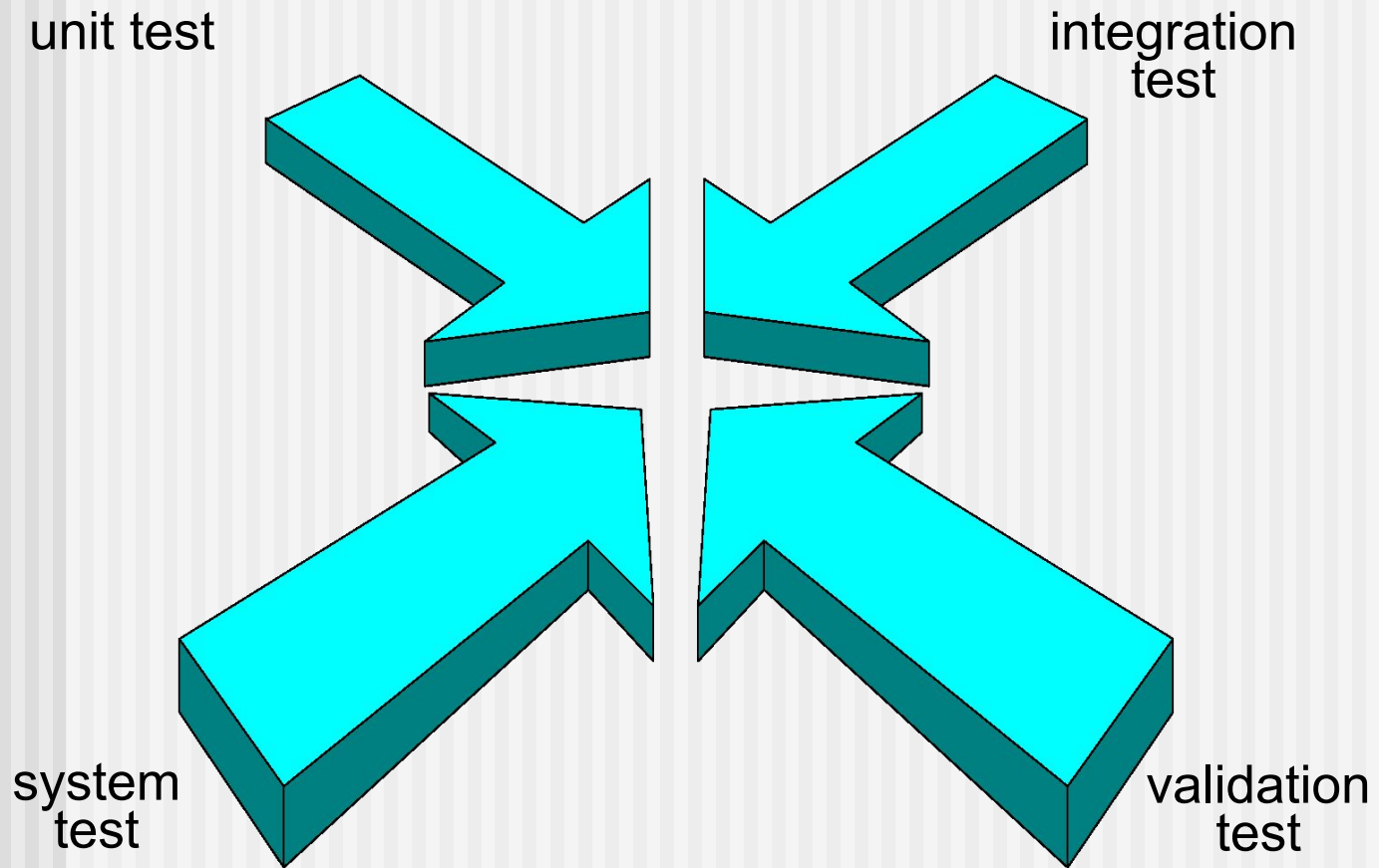
There are 10^{14} possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!



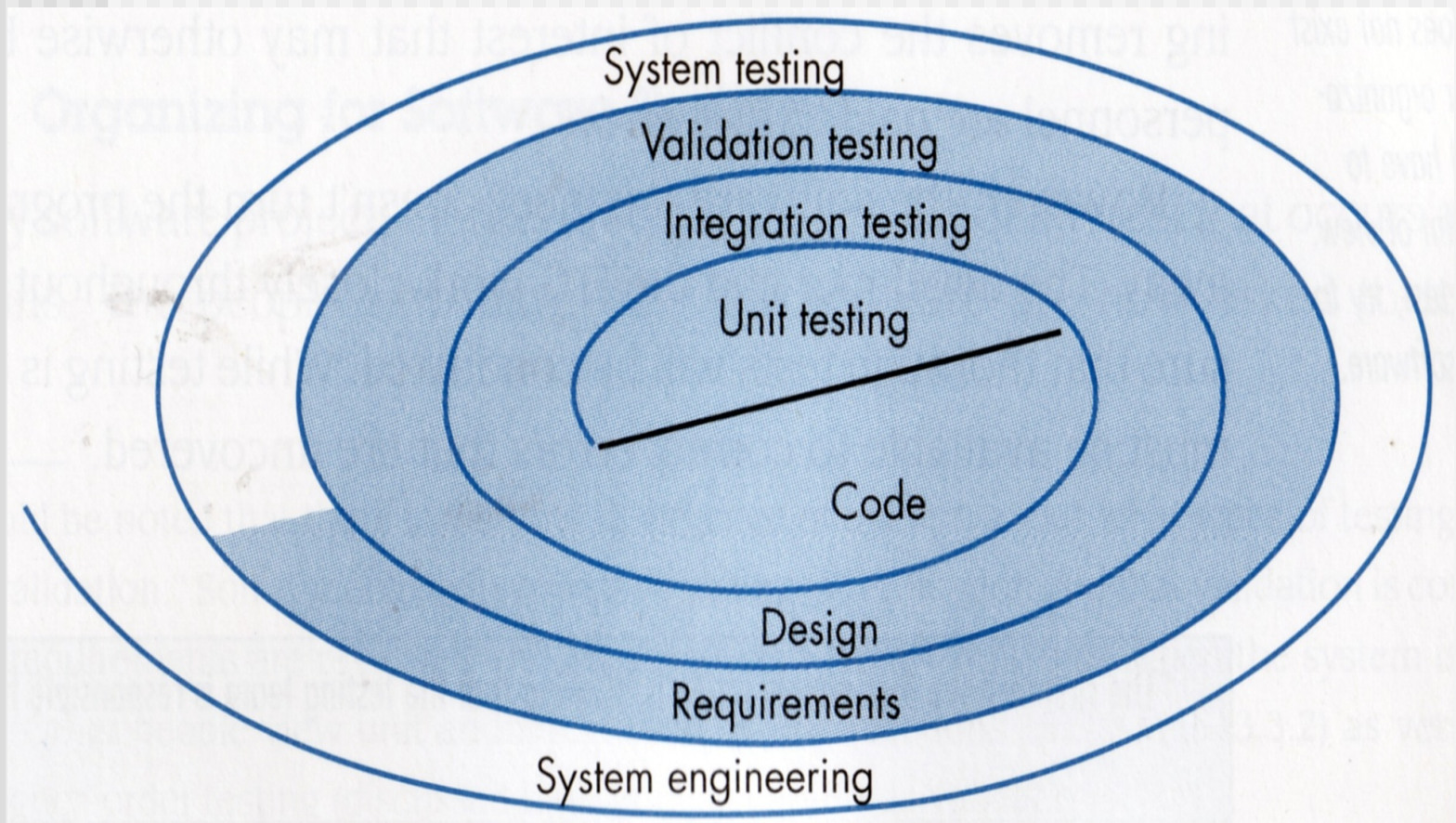
Software Testing



Testing Strategy



Testing Strategy

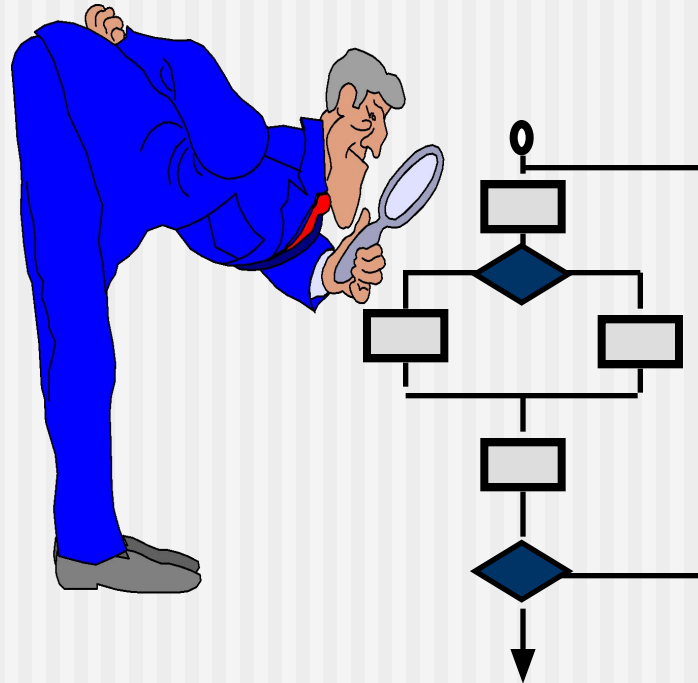


White-box Testing

- Uses the control structure part of component-level design to derive the test cases
- These test cases
 - Guarantee that all independent paths within a module have been exercised at least once
 - Exercise all logical decisions on their true and false sides
 - Execute all loops at their boundaries and within their operational bounds
 - Exercise internal data structures to ensure their validity

“Bugs lurk in corners and congregate at boundaries”

White-Box Testing



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

Why Cover?

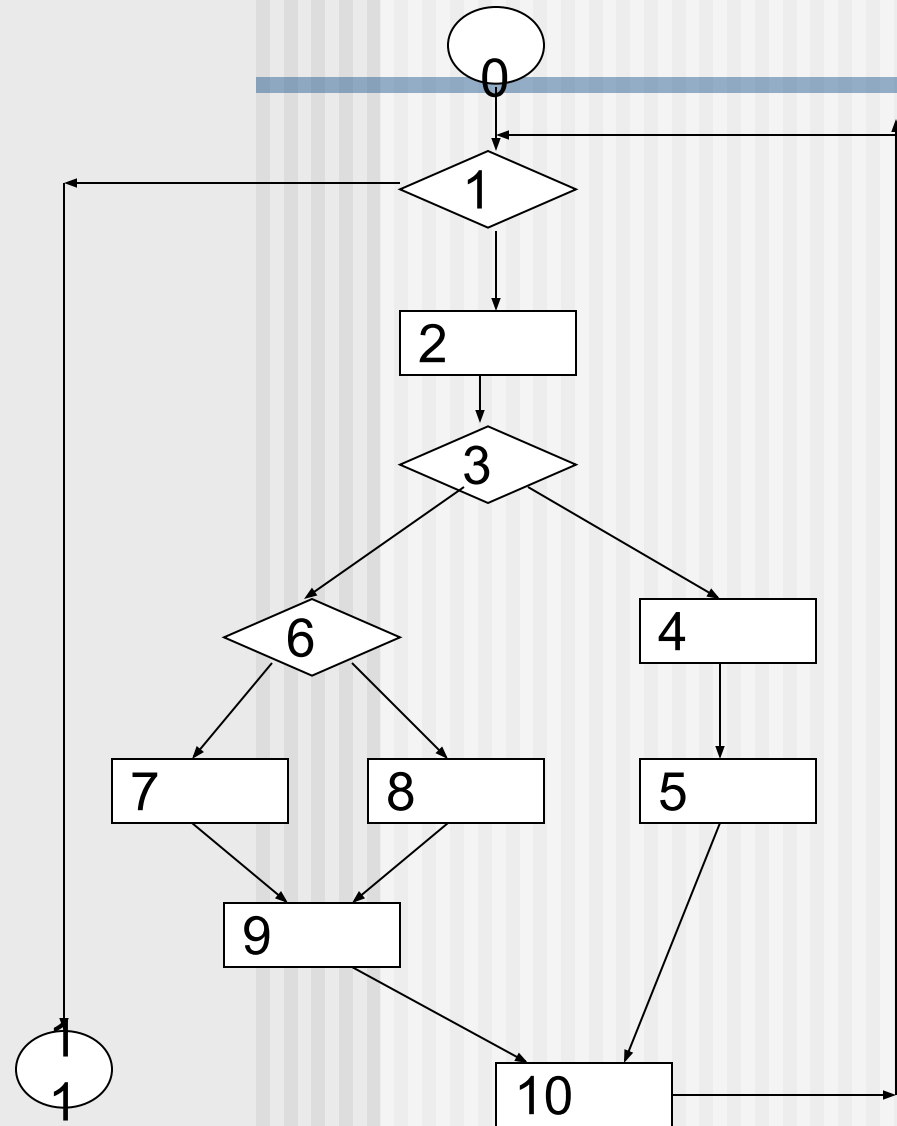
- **logic errors and incorrect assumptions are inversely proportional to a path's execution probability**
- **we often believe that a path is not likely to be executed; in fact, reality is often counter intuitive**
- **typographical errors are random; it's likely that untested paths will contain some**

BPT: Flow Graph Notation

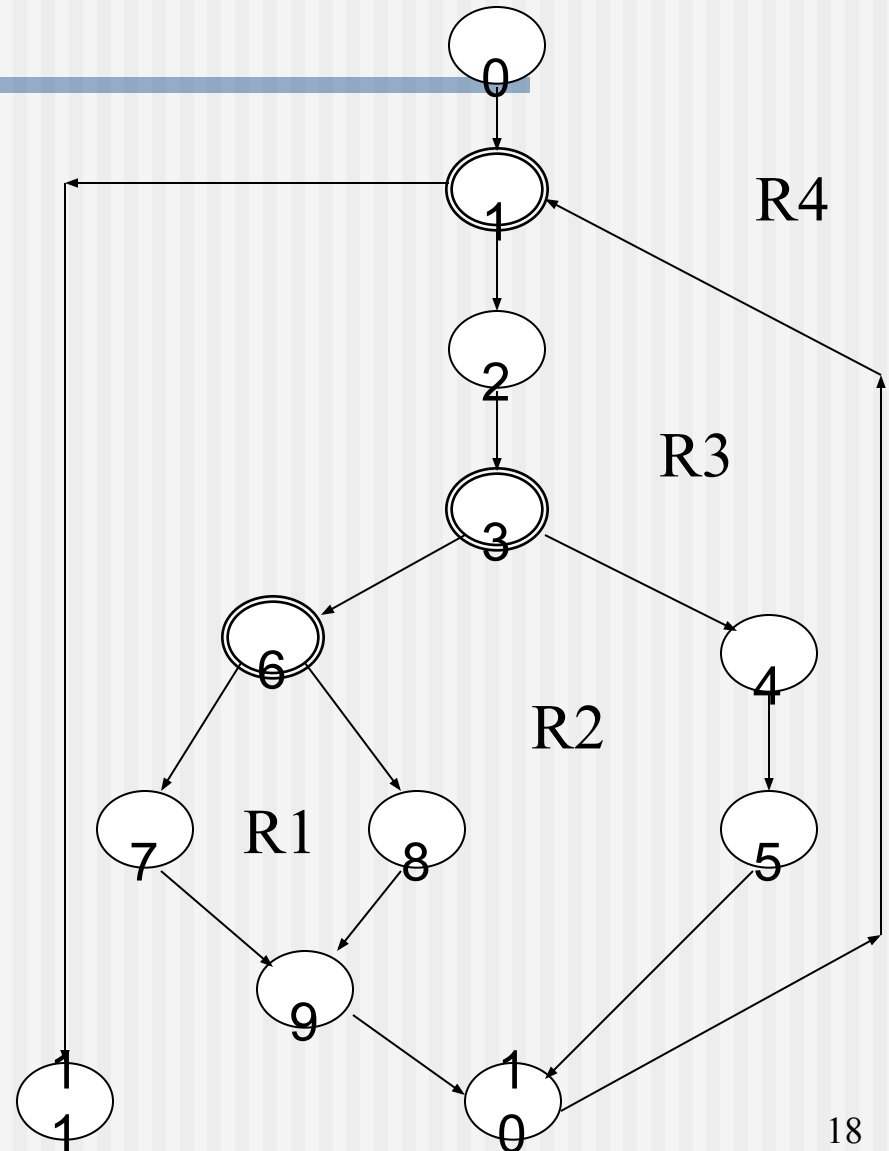
- A circle in a graph represents a node, which stands for a sequence of one or more procedural statements
- A node containing a simple conditional expression is referred to as a predicate node
 - Each compound condition in a conditional expression containing one or more Boolean operators (e.g., and, or) is represented by a separate predicate node
 - A predicate node has two edges leading out from it (True and False)
- An edge, or a link, is a an arrow representing flow of control in a specific direction
 - An edge must start and terminate at a node
 - An edge does not intersect or cross over another edge
- Areas bounded by a set of edges and nodes are called regions
- When counting regions, include the area outside the graph as a region, too

BPT: Flow Graph Example

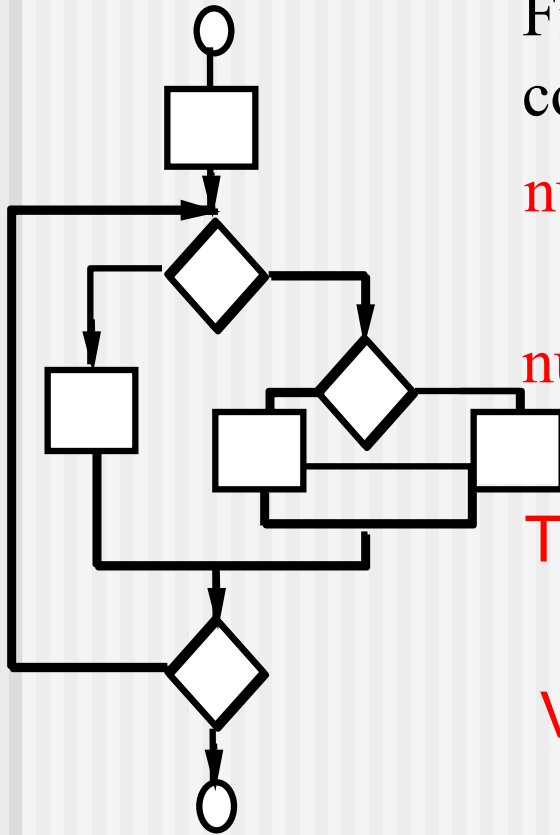
FLOW CHART



FLOW GRAPH



Basis Path Testing



First, we compute the cyclomatic complexity:

number of simple decisions + 1

or

number of enclosed areas + 1

or

The number of regions

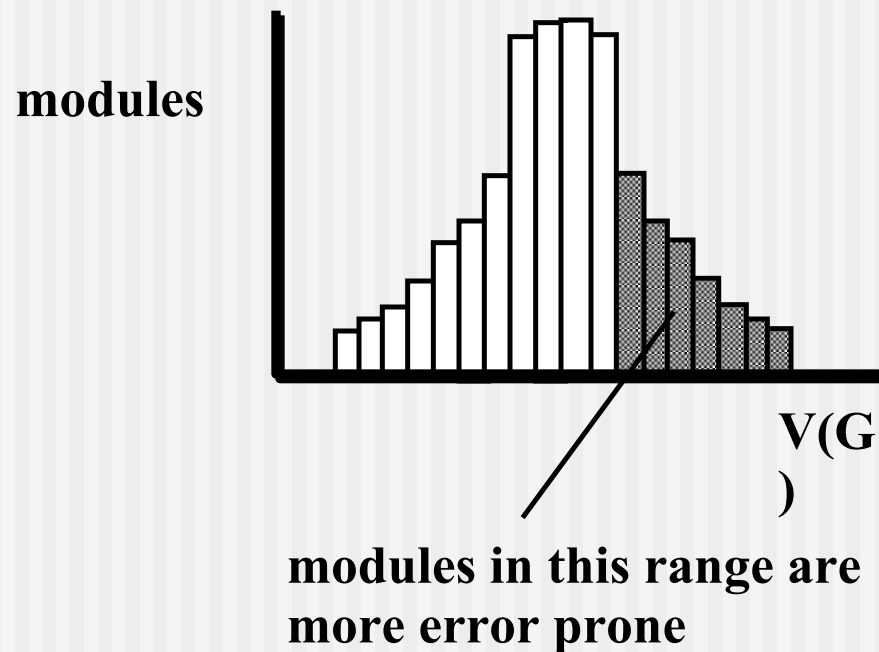
or

$$V(G) = E - N + 2$$

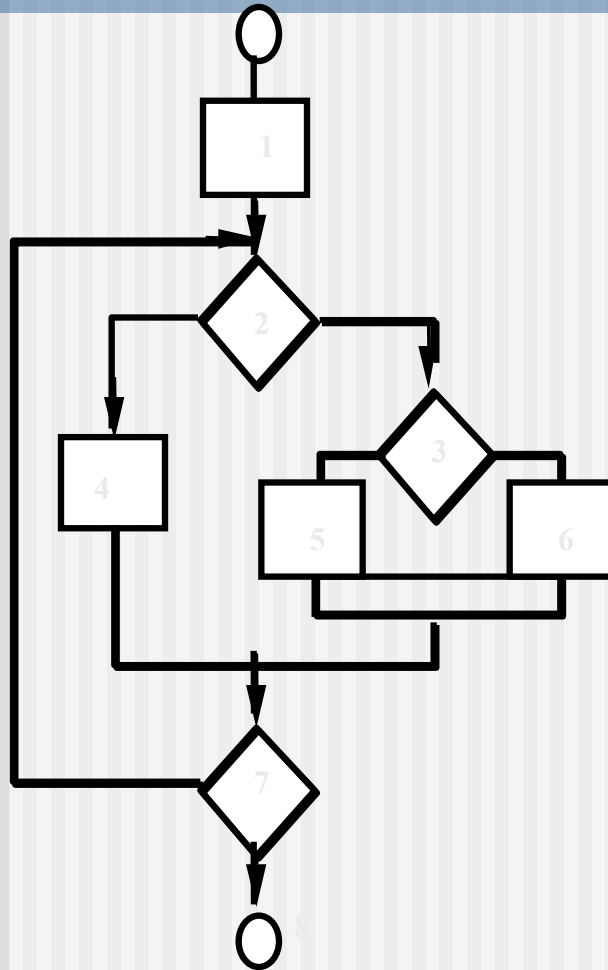
In this case, $V(G) = 4$

Cyclomatic Complexity

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

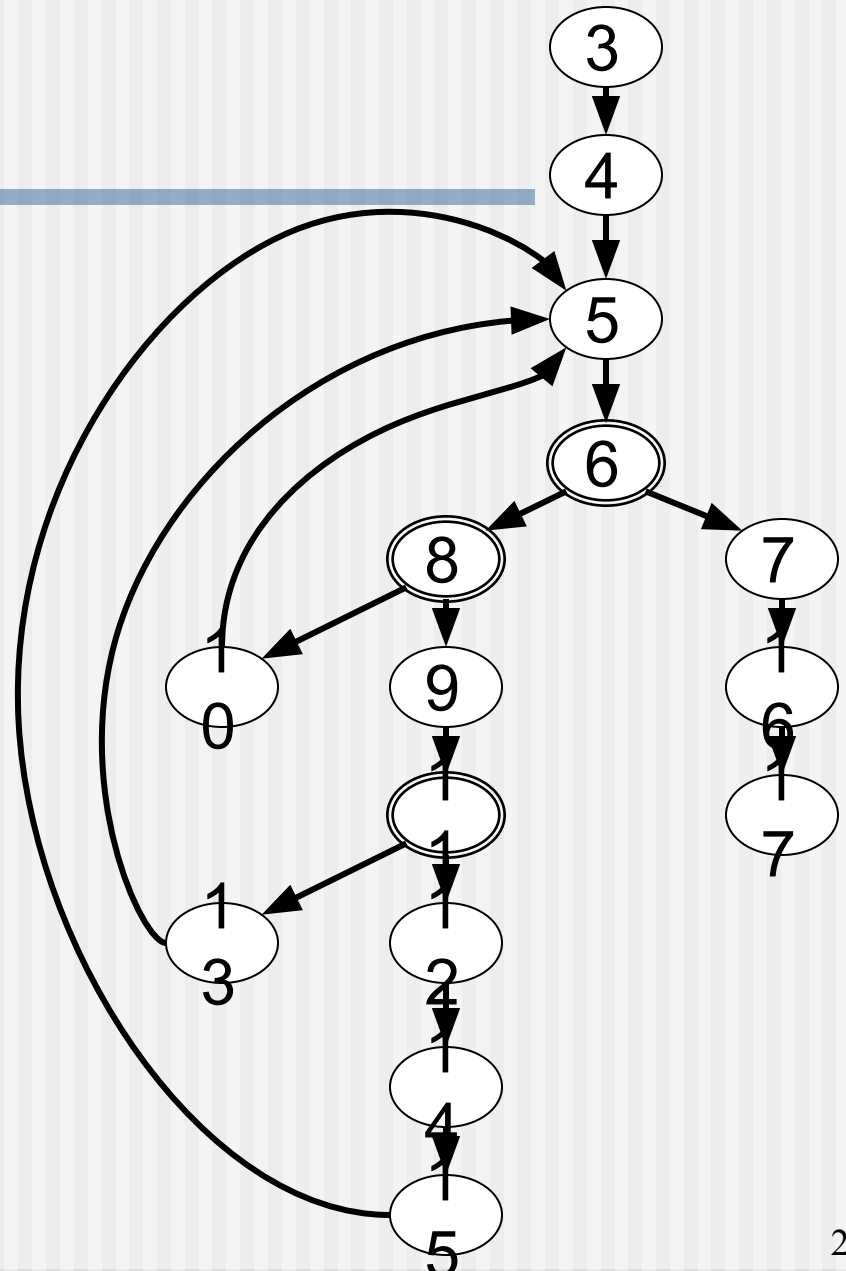
Path 3: 1,2,4,7,8

Path 4: 1,2,4,7,2,4,...7,8

Finally, we derive test cases to exercise these paths.

A Second Flow Graph Example

```
1  int functionY(void)
2  {
3      int x = 0;
4      int y = 19;
5  A: x++;
6      if (x > 999)
7          goto D;
8      if (x % 11 == 0)
9          goto B;
10     else goto A;
11 B: if (x % y == 0)
12     goto C;
13     else goto A;
14 C: printf("%d\n", x);
15     goto A;
16 D: printf("End of list\n");
17     return 0;
18 }
```



A Sample Function to Diagram and Analyze

```
1  int functionZ(int y)
2  {
3  int x = 0;

4  while (x <= (y * y))
5      {
6          if ((x % 11 == 0) &&
7              (x % y == 0))
8              {
9                  printf("%d", x);
10                 x++;
11             } // End if
12         else if ((x % 7 == 0) ||
13                 (x % y == 1))
14             {
15                 printf("%d", y);
16                 x = x + 2;
17             } // End else
18         printf("\n");
19     } // End while

20 printf("End of list\n");
21 return 0;
22 } // End functionZ
```

A Sample Function to Diagram and Analyze

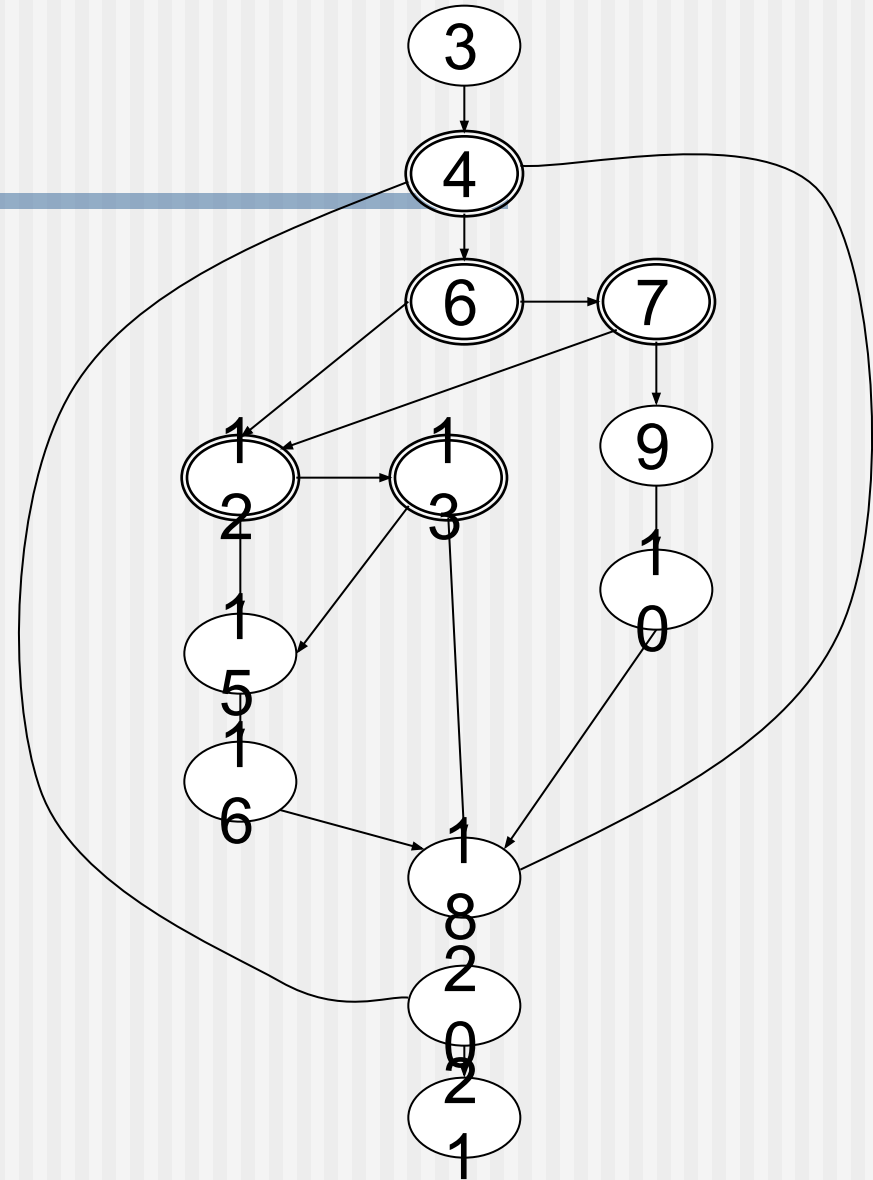
```

1  int functionZ(int y)
2  {
3  int x = 0;

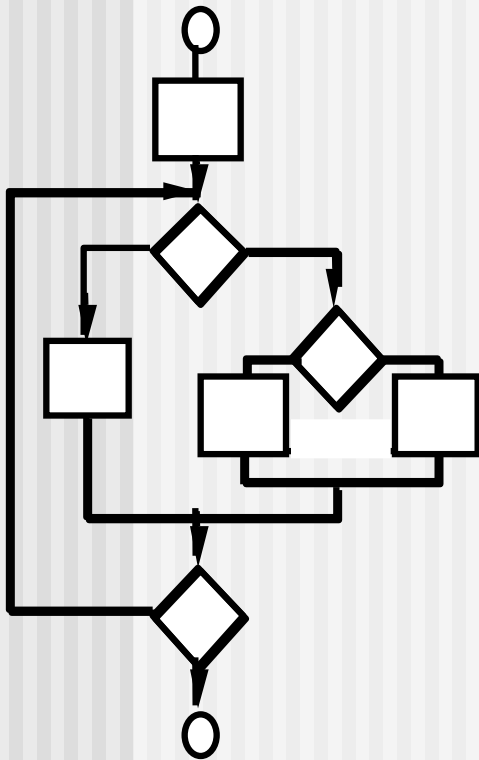
4  while (x <= (y * y))
5  {
6      if ((x % 11 == 0) &&
7          (x % y == 0))
8          {
9              printf("%d", x);
10             x++;
11         } // End if
12     else if ((x % 7 == 0) ||
13              (x % y == 1))
14         {
15             printf("%d", y);
16             x = x + 2;
17         } // End else
18     printf("\n");
19 } // End while

20 printf("End of list\n");
21 return 0;
22 } // End functionZ

```



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.

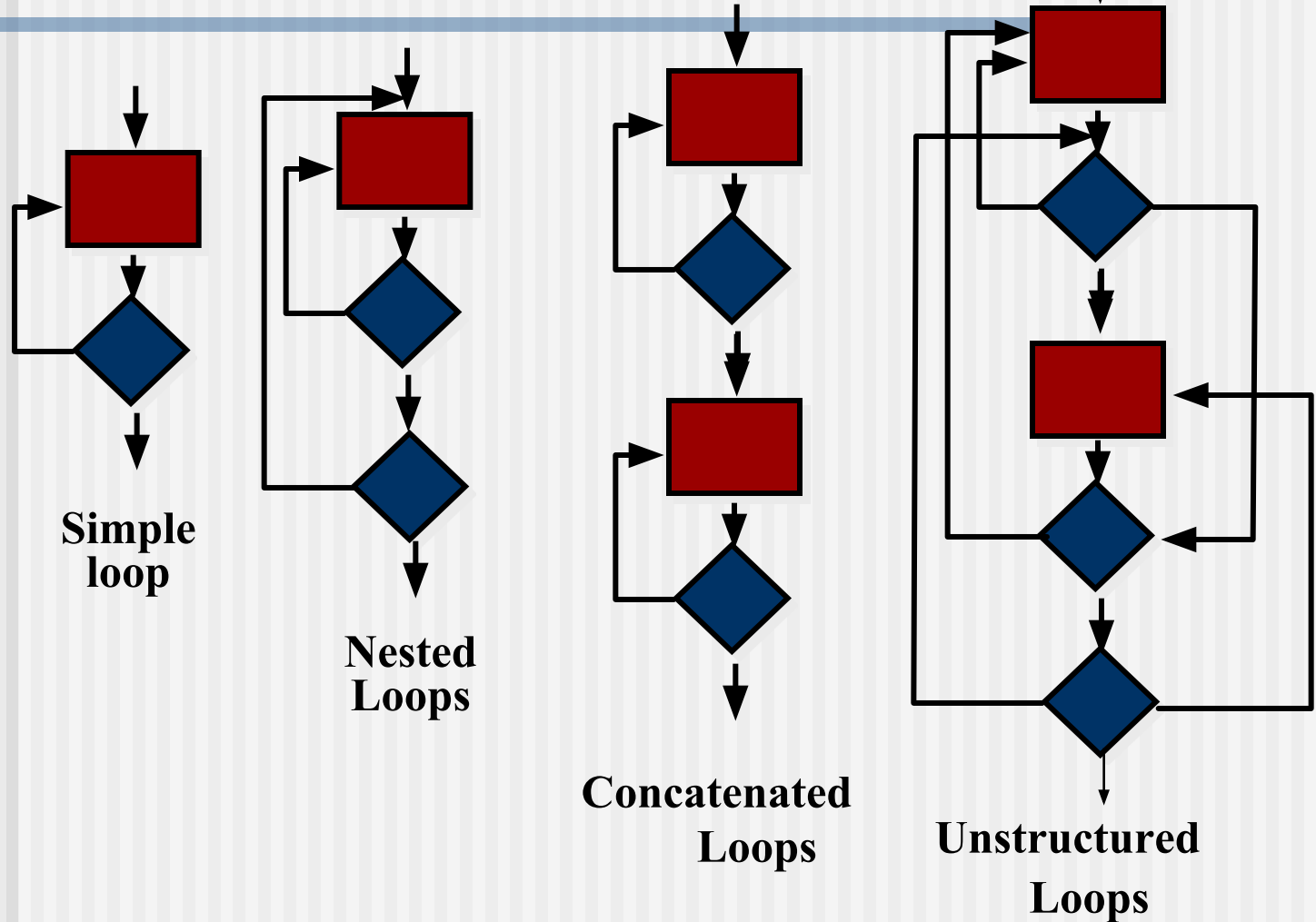
Graph Matrices

- 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

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.

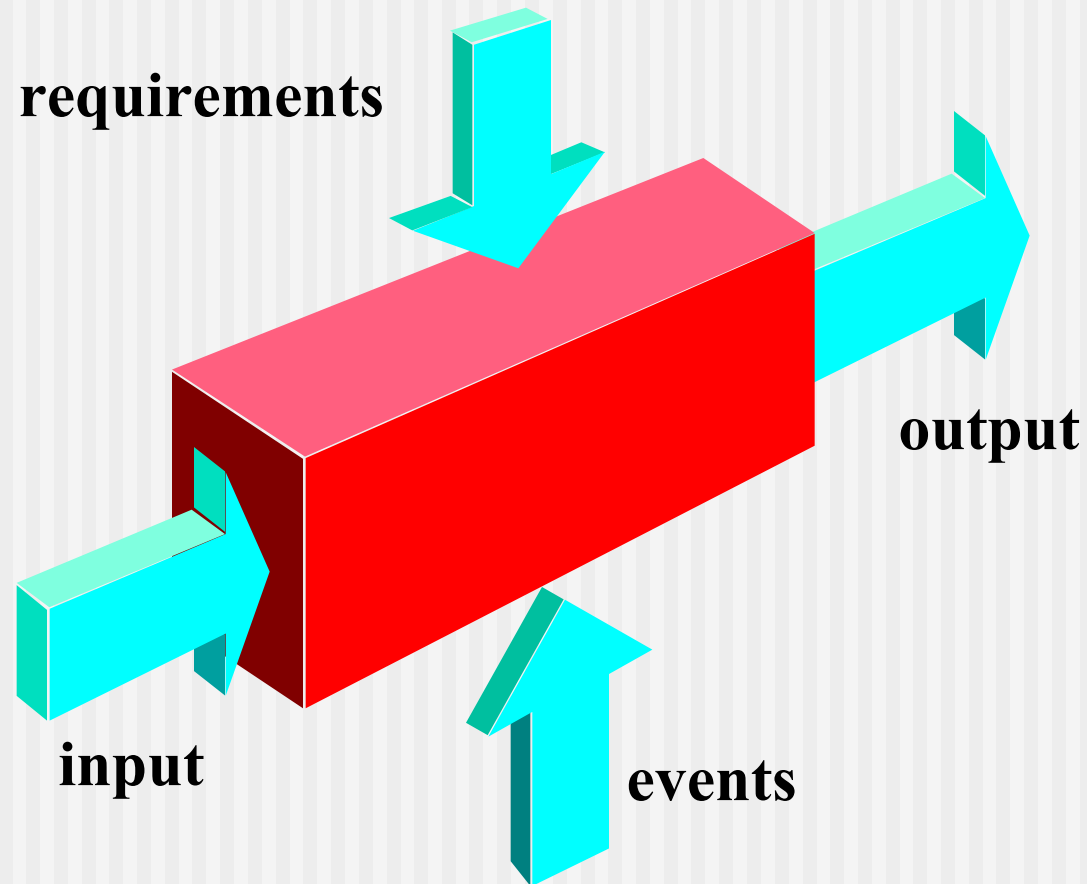
Move out one loop and set it up as in step 2, holding all other loops at typical values. Continue this step until the outermost loop has been tested.

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 Categories

- Incorrect or missing functions
- Interface errors
- Errors in data structures or external data base access
- Behavior or performance errors
- Initialization and termination errors

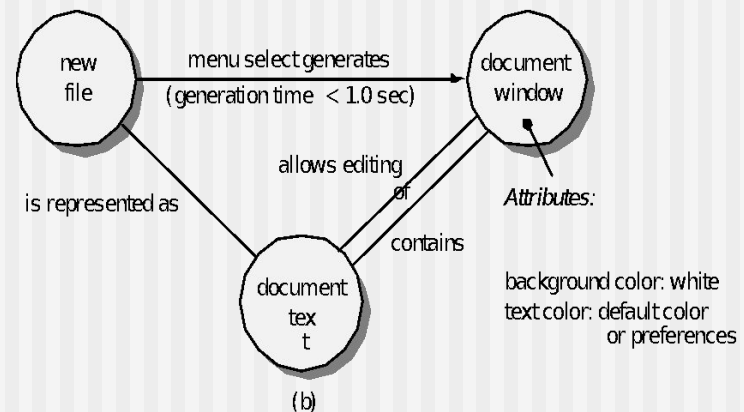
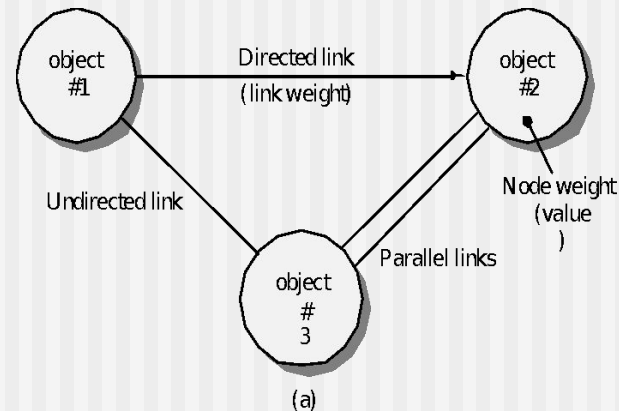
Black-Box Testing

- 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

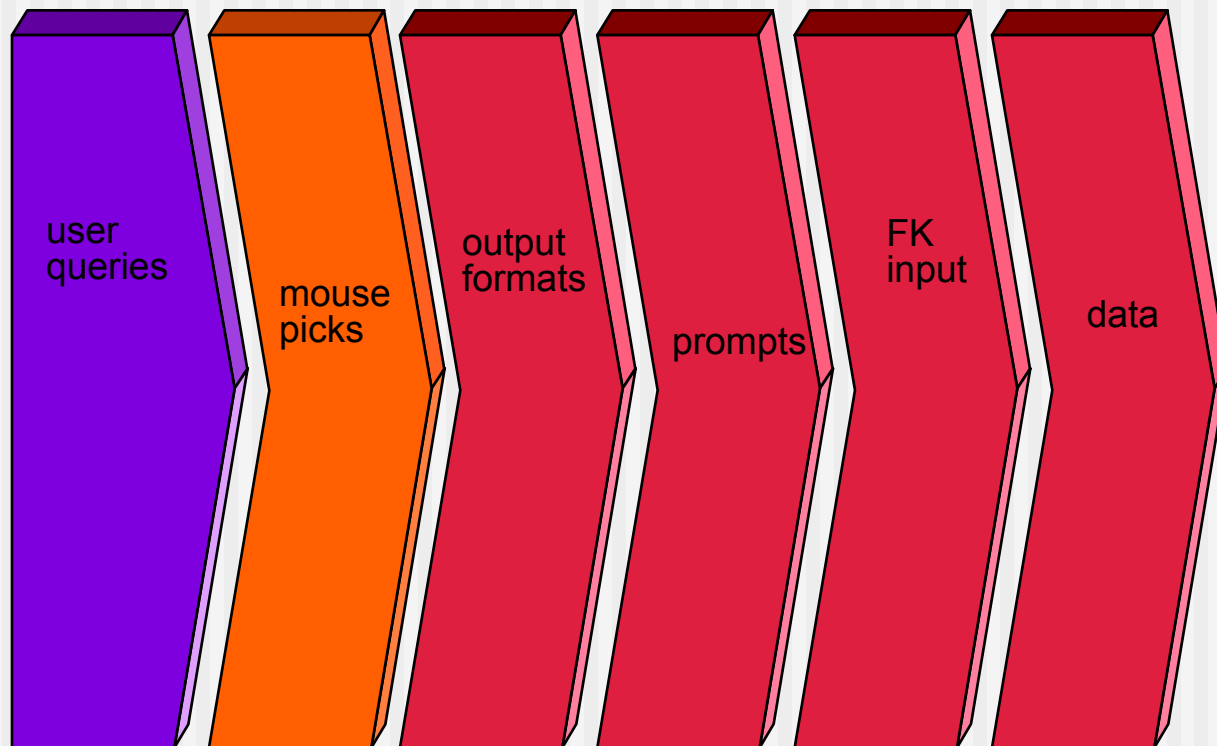
In this context, we consider the term “objects” in the broadest possible context. It encompasses data objects, traditional components (modules), and object-oriented elements of computer software.



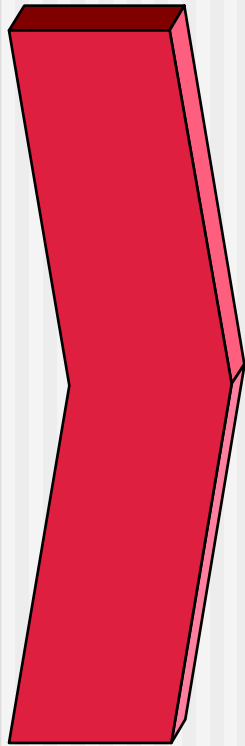
Equivalence Partitioning

- A black-box testing method that divides the input domain of a program into classes of data from which test cases are derived
- An ideal test case single-handedly uncovers a complete class of errors, thereby reducing the total number of test cases that must be developed
- Test case design is based on an evaluation of equivalence classes for an input condition
- An equivalence class represents a set of valid or invalid states for input conditions
- From each equivalence class, test cases are selected so that the largest number of attributes of an equivalence class are exercised at once

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

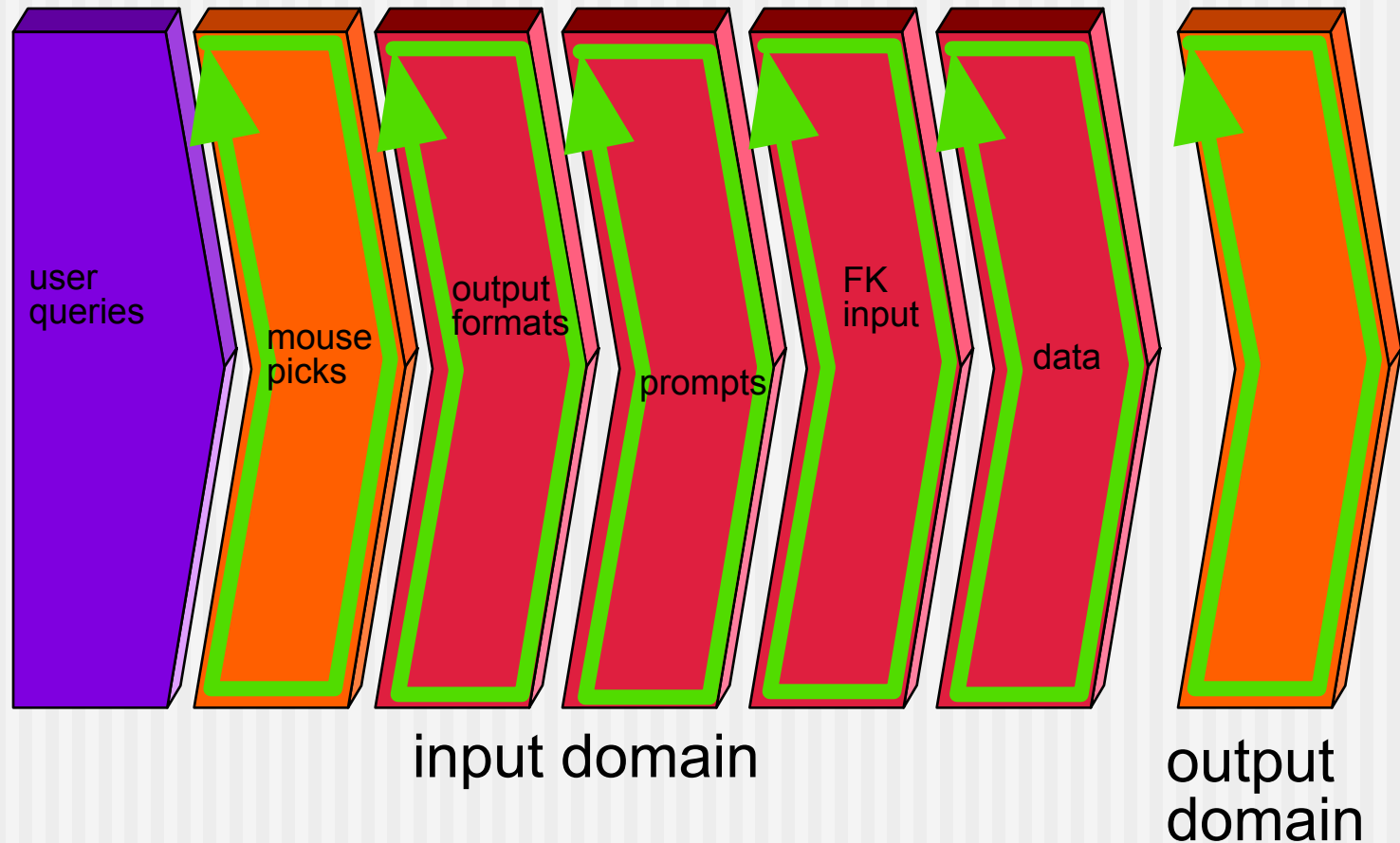
Guidelines for Defining Equivalence Classes

- If an input condition specifies a range, one valid and two invalid equivalence classes are defined
 - Input range: 1 – 10 Eq classes: {1..10}, {x < 1}, {x > 10}
- If an input condition requires a specific value, one valid and two invalid equivalence classes are defined
 - Input value: 250 Eq classes: {250}, {x < 250}, {x > 250}
- If an input condition specifies a member of a set, one valid and one invalid equivalence class are defined
 - Input set: {-2.5, 7.3, 8.4} Eq classes: {-2.5, 7.3, 8.4}, {any other x}
- If an input condition is a Boolean value, one valid and one invalid class are defined
 - Input: {true condition} Eq classes: {true condition}, {false condition}

Boundary Value Analysis

- A greater number of errors occur at the boundaries of the input domain rather than in the "center"
- Boundary value analysis is a test case design method that complements equivalence partitioning
 - It selects test cases at the edges of a class
 - It derives test cases from both the input domain and output domain

Boundary Value Analysis

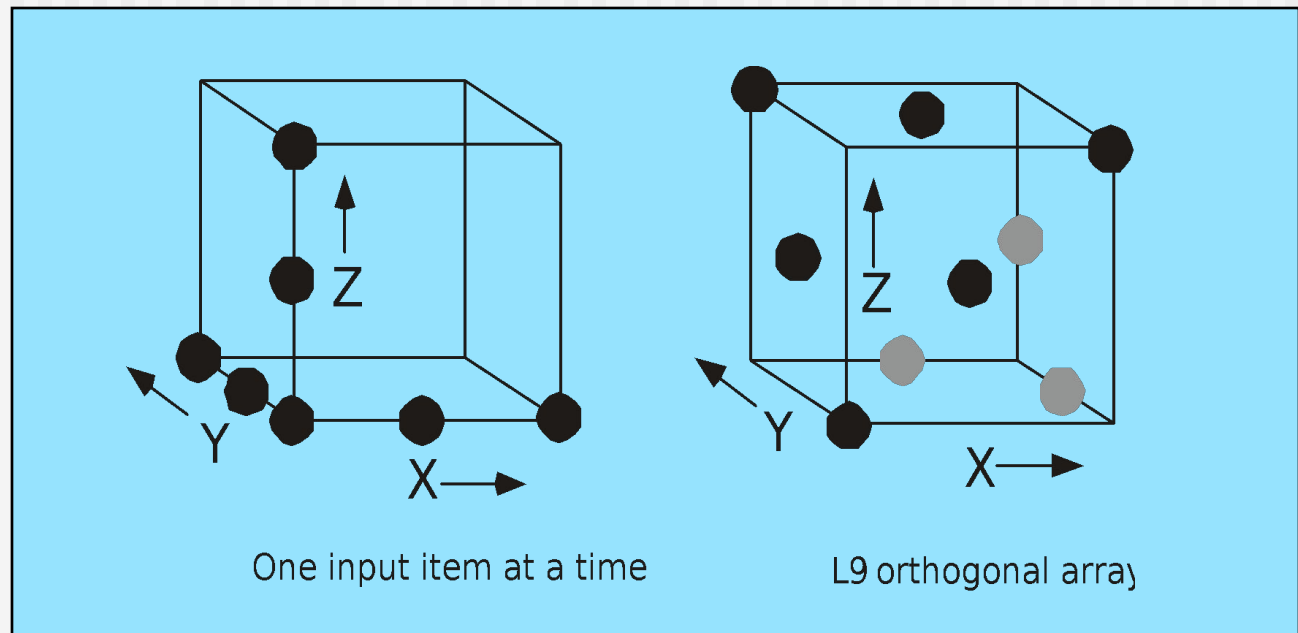


Guidelines for Boundary Value Analysis

- 1. If an input condition specifies a range bounded by values *a* and *b*, test cases should be designed with values *a* and *b* as well as values just above and just below *a* and *b*
- 2. If an input condition specifies a number of values, test case should be developed that exercise the **minimum** and **maximum** numbers. Values just above and just below the **minimum** and **maximum** are also tested
- 3. Apply guidelines 1 and 2 to output conditions; produce output that reflects the **minimum** and the **maximum** values expected; also test the values just below and just above
- 4. If internal program data structures have prescribed boundaries (e.g., an array), design a test case to exercise the data structure at its **minimum** and **maximum** boundaries

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



Debugging

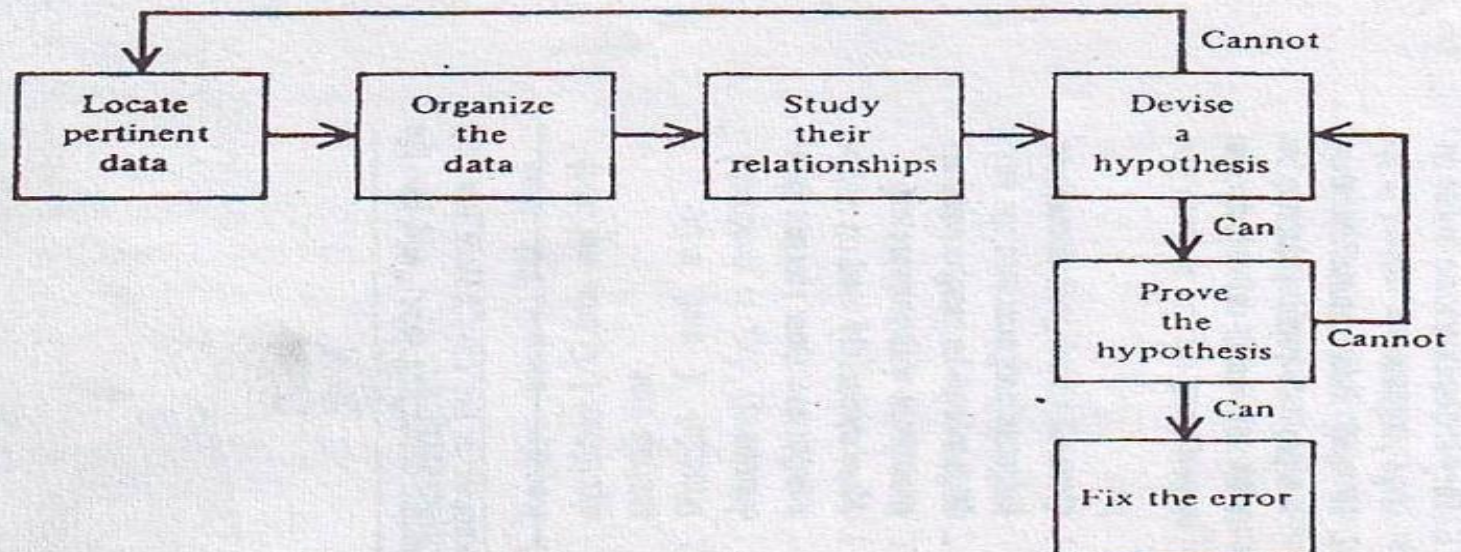
- Once the error has been discovered, begin an investigation to localize the error,
- i.e., to find out which module or interface is causing it.
- Then, this section of code is studied to determine the cause of the problem.
- This process is called **debugging**.
- It generally involves further specialized testing and code reading to pinpoint the source of the error.

Debugging Strategies

- **Trial and Error**
- **Backtracking**
- **Forward Tracking**

The Inductive Debugging Process

- The inductive approach comes from the formulation of a single working hypothesis based on:
 - the data,
 - the analysis of existing data,
 - and on specially collected data to prove or disprove the working hypothesis.



The Deductive Debugging Process

- The process of deduction begins by enumerating all causes or hypotheses which seem possible.
- Then, one by one, particular causes are ruled out until a single one remains for validation.

