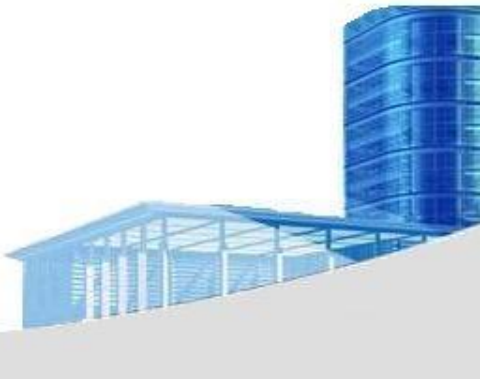
Ch.23 Testing Conventional Applications



- **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

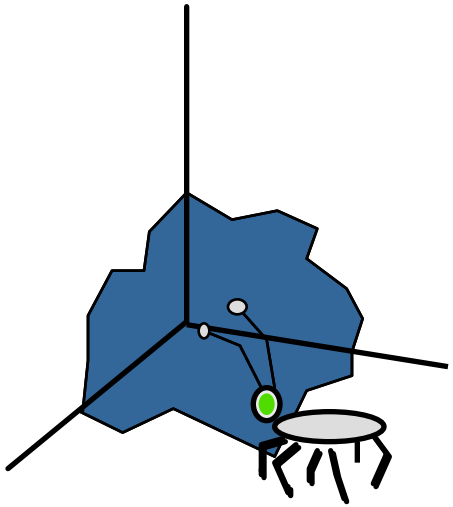
- **Internal and External Views**

  - 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

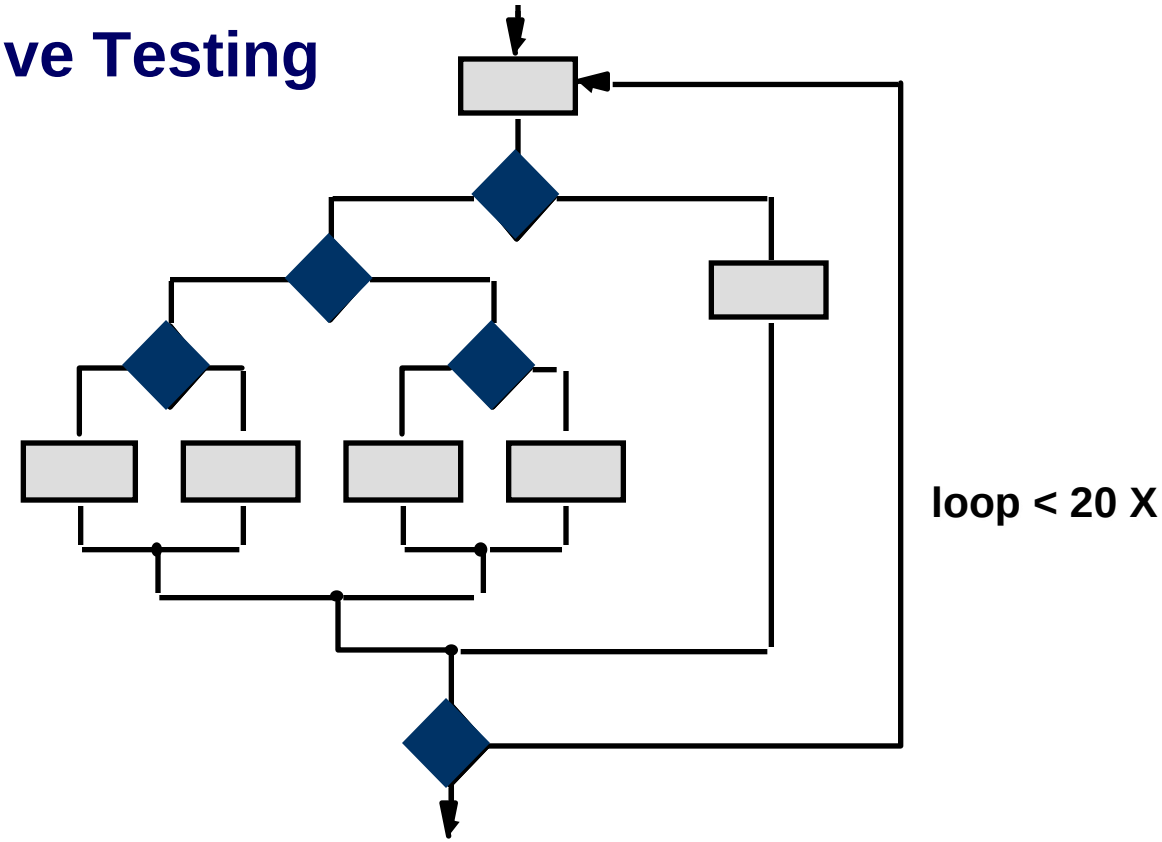


OBJECTIVE **to uncover errors**

CRITERIA **in a complete manner**

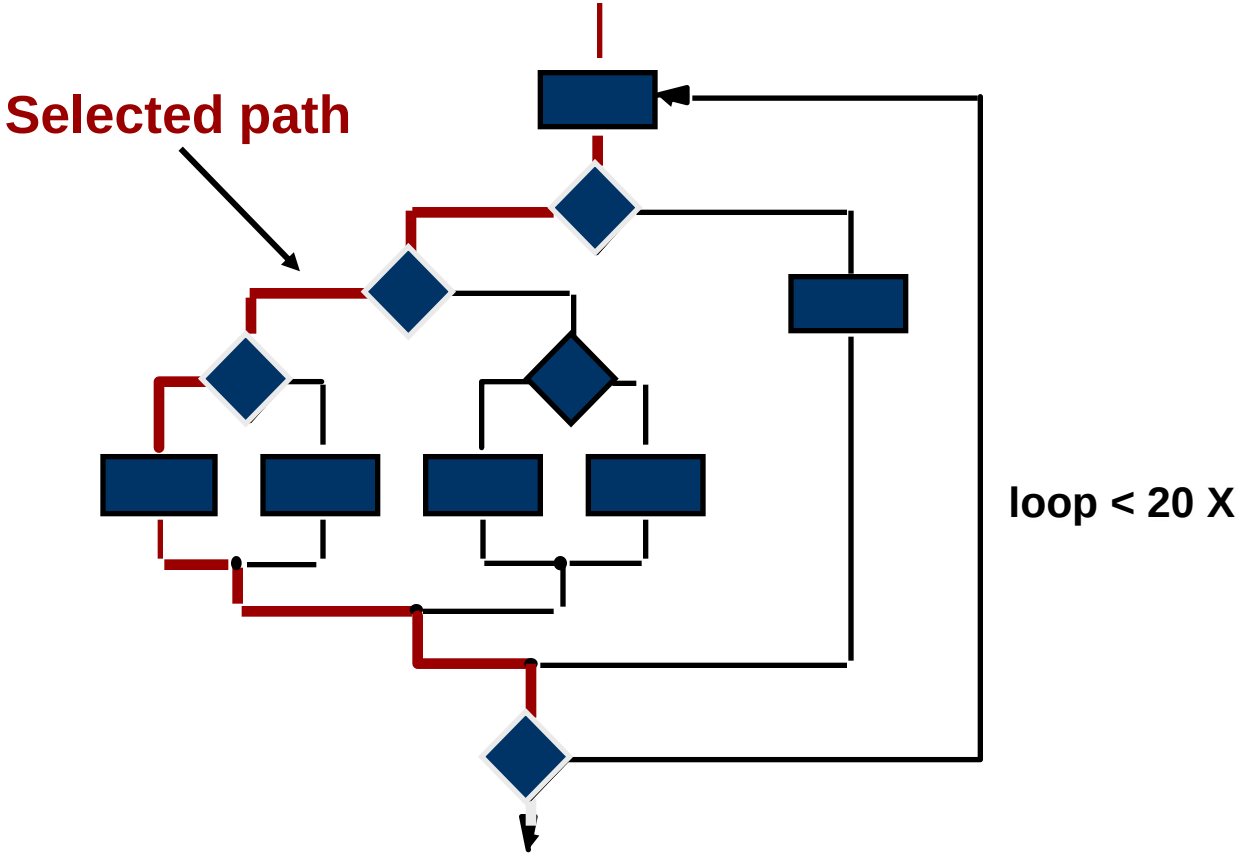
CONSTRAINT **with a minimum of effort and time**

- **Exhaustive Testing**

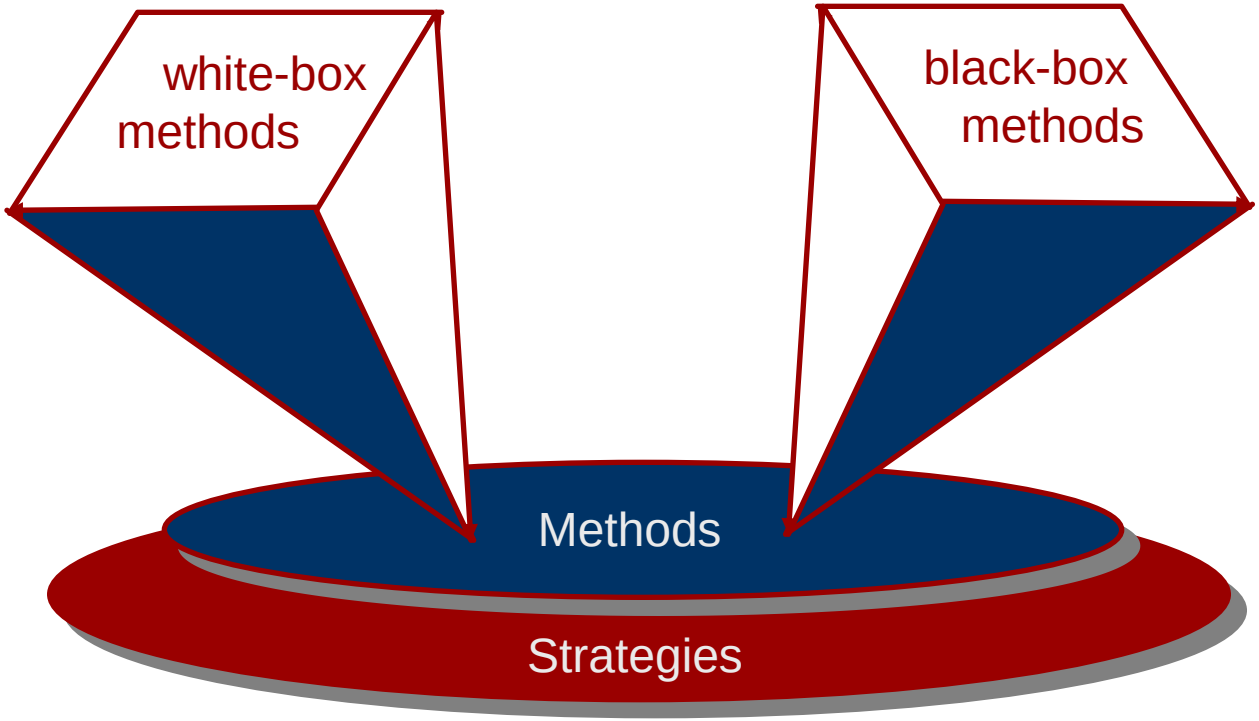


There are 10^{14} possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!

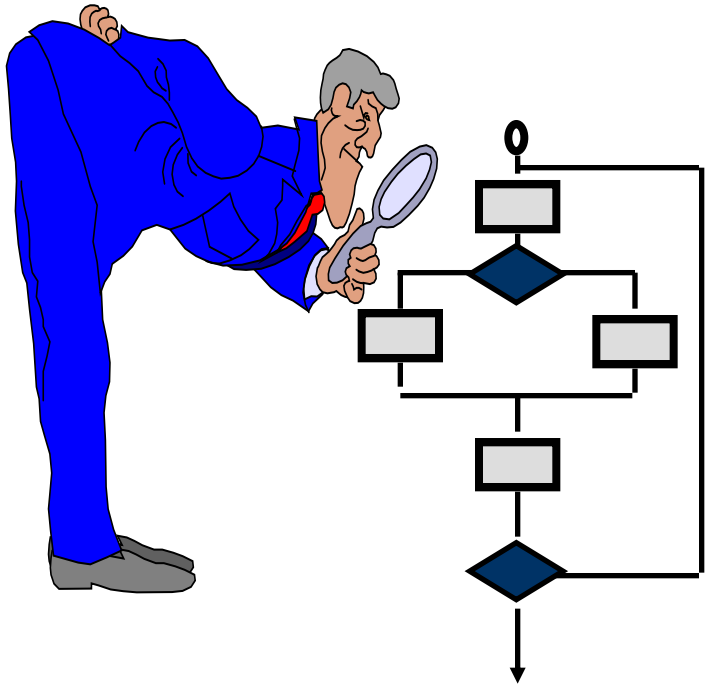
- **Selective Testing**



- **Software Testing**



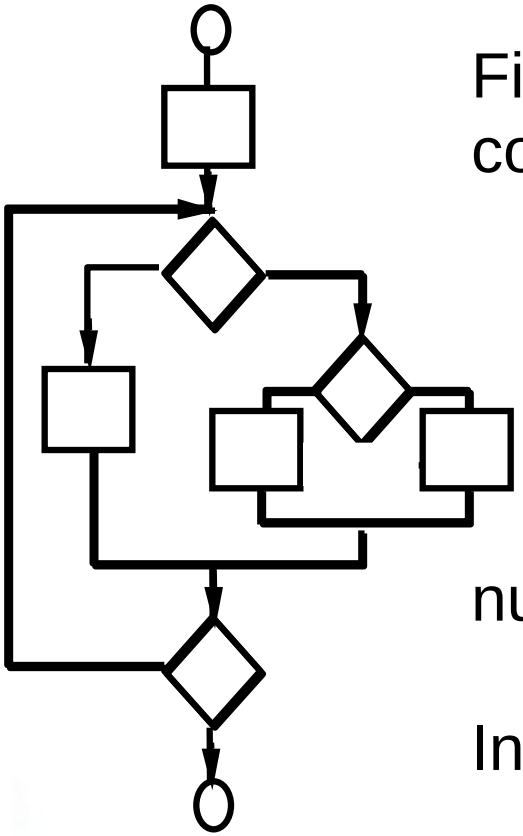
- **White-Box Testing**



... 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

- **Basis Path Testing**



First, we compute the cyclomatic complexity:

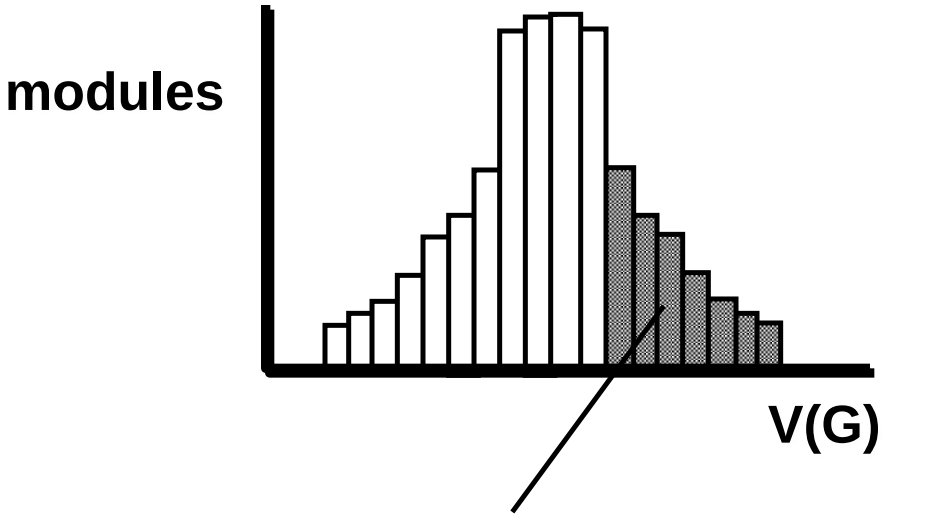
or

number of enclosed areas + 1

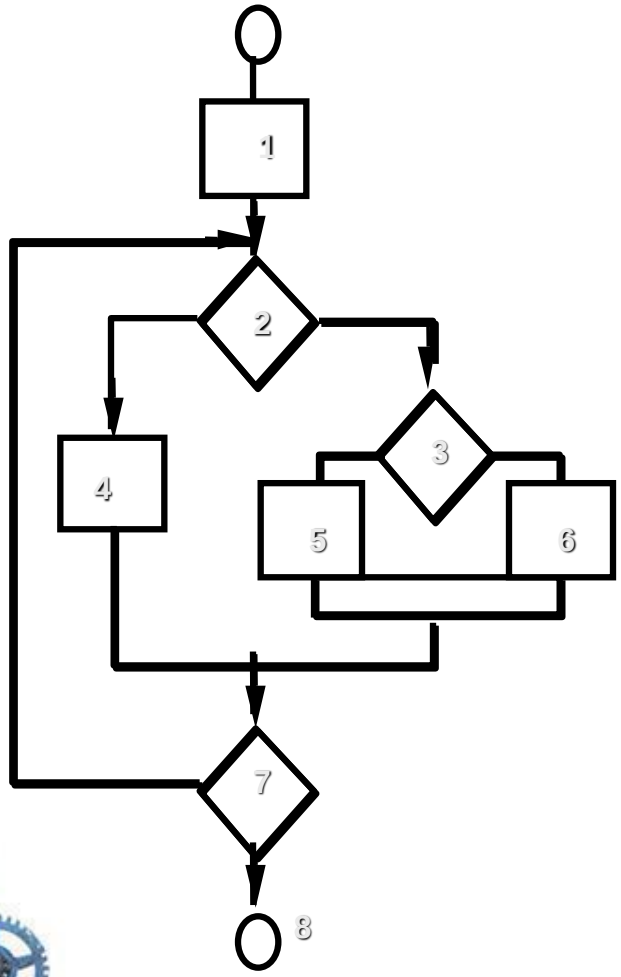
In this case, V(G) = 4

- **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

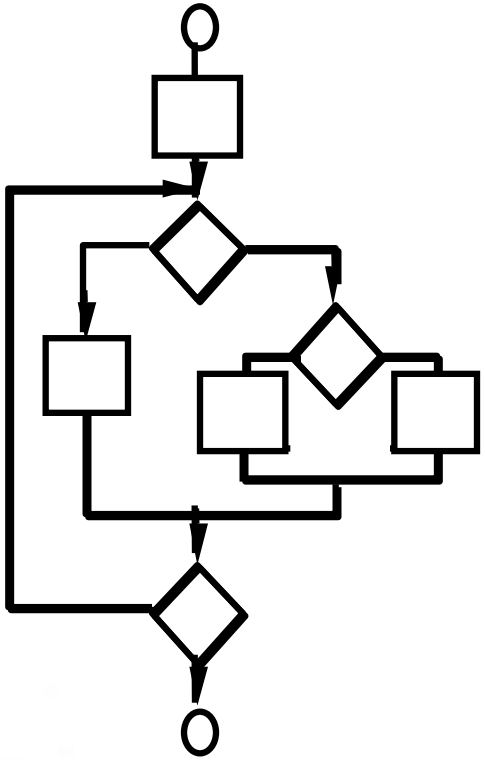
Path 2: 1,2,3,5,7,8

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.

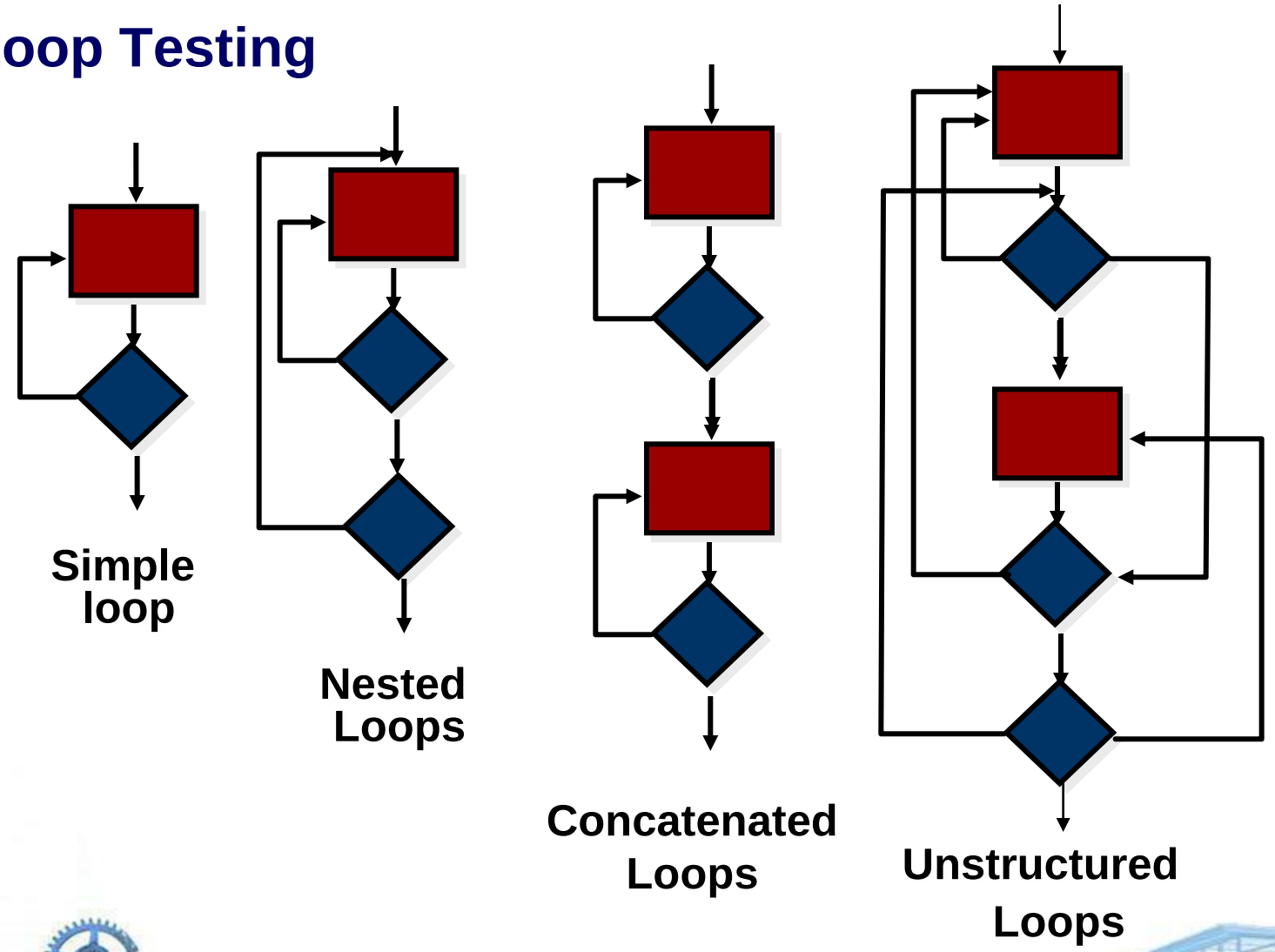
- **Graph Matrices**
  - A graph matrix is a square matrix whose size (i.e., number of rows and columns) is equal to the number of nodes on a flow graph
  - Each row and column corresponds to an identified node, and matrix entries correspond to connections (an edge) between nodes.
  - By adding a link 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**
  - 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

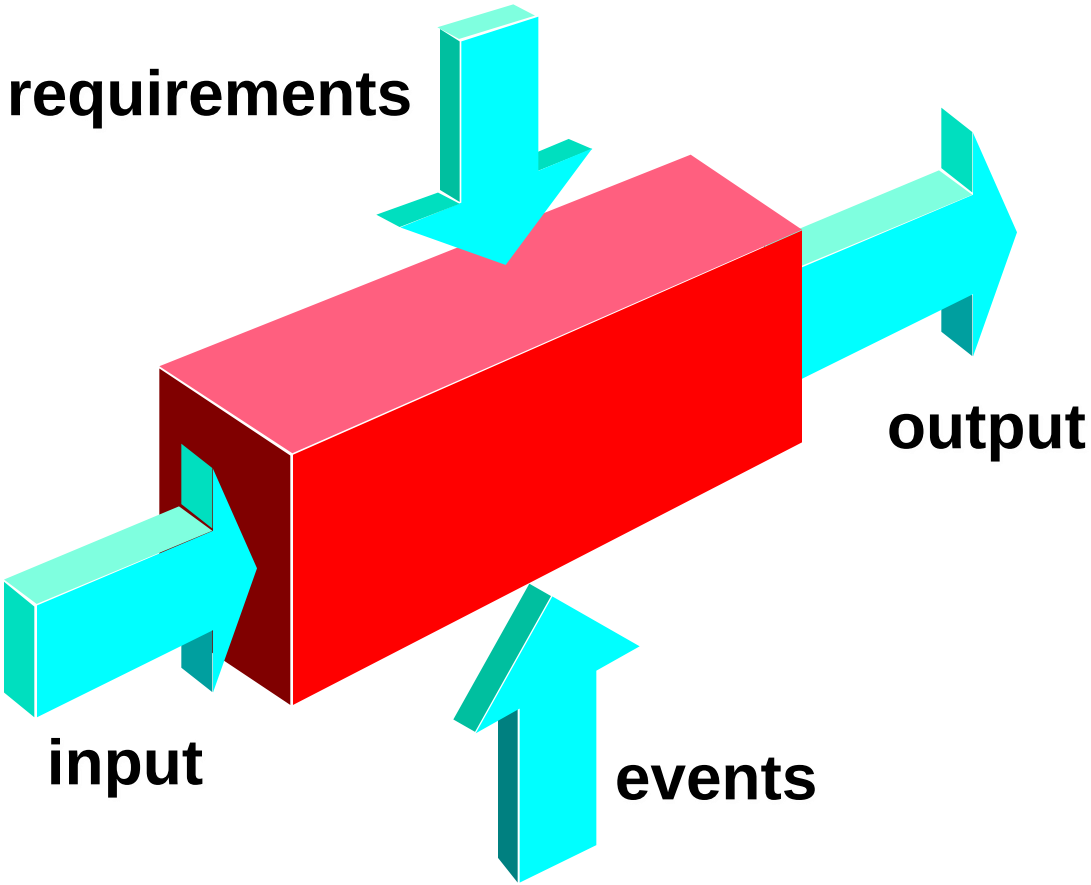
      endif*
    - for example, the final loop counter value of loop 1 is used to initialize loop 2.

- **Loop Testing: Nested 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

    where n is the maximum number of allowable passes

- **Black-Box Testing**

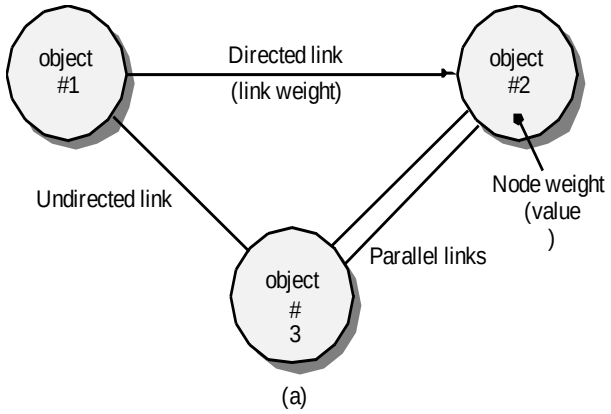


- **Black-Box Testing**
  - How is functional validity tested?
  - How is system behavior and performance tested?
  - What classes of input will make good test cases?
  - Is the system particularly sensitive to certain input values?
  - How are the boundaries of a data class isolated?
  - What data rates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

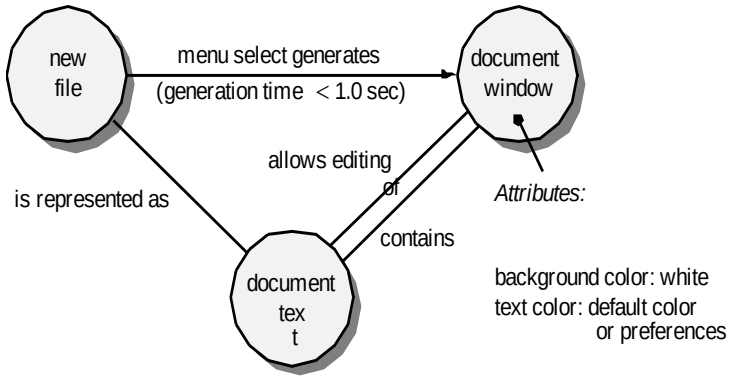
- **Graph-Based Methods**

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.

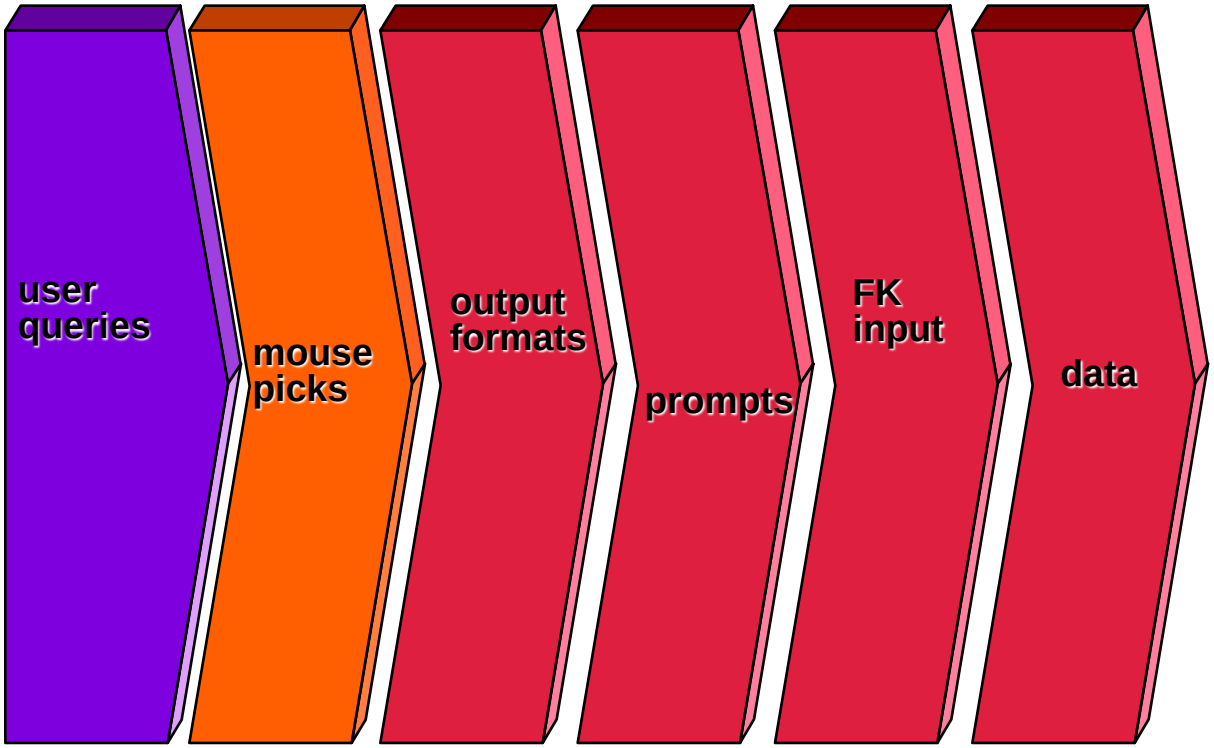


(a)

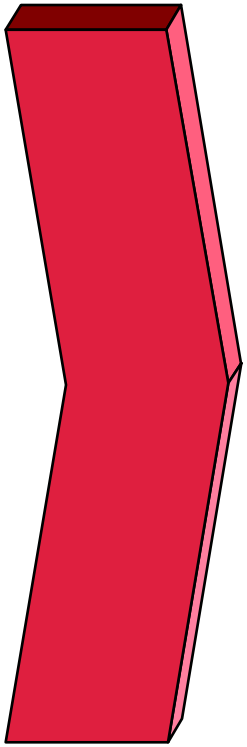


(b)

- **Equivalence Partitioning**



- **Sample Equivalence Classes**



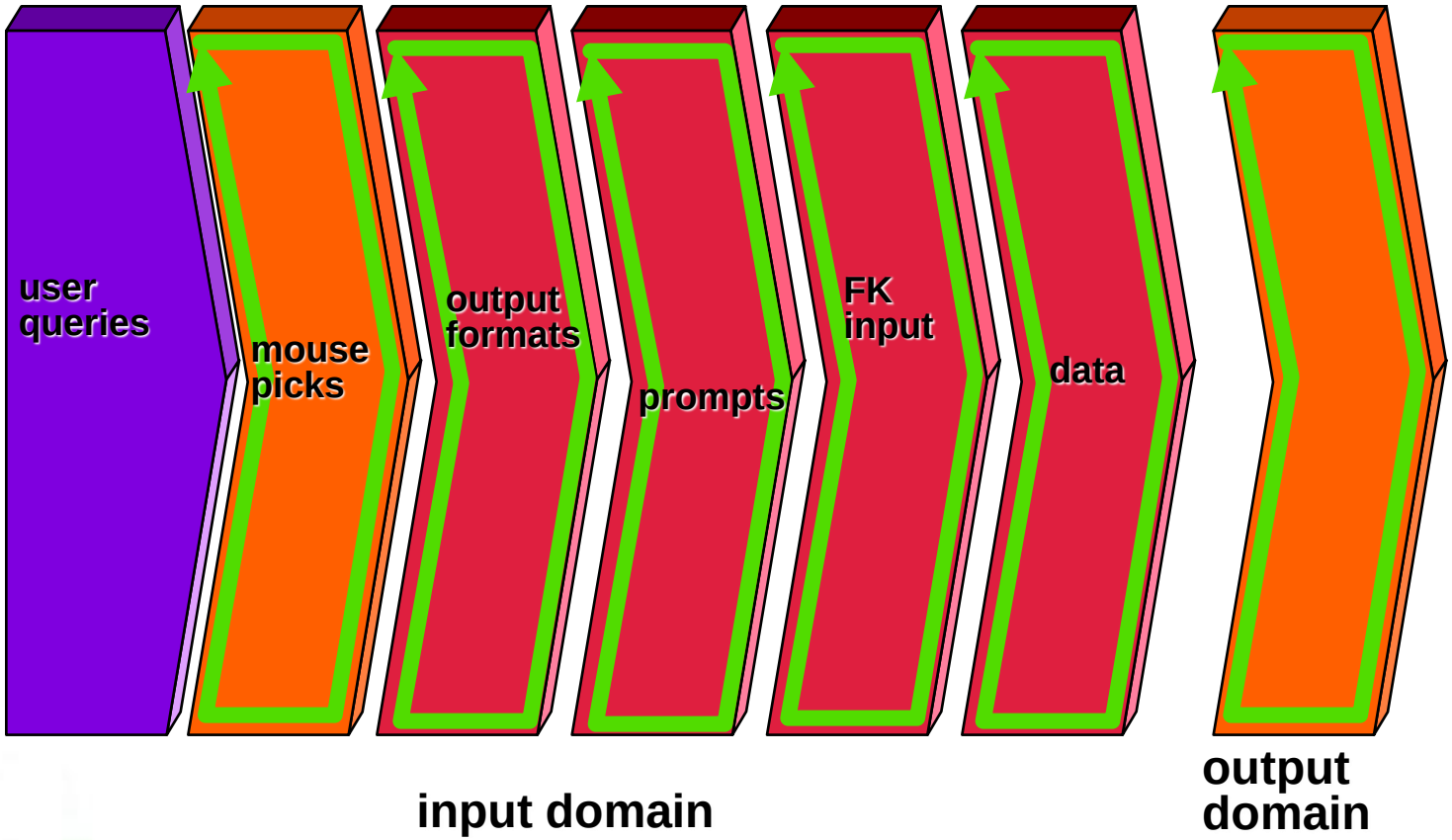
Valid data

- 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

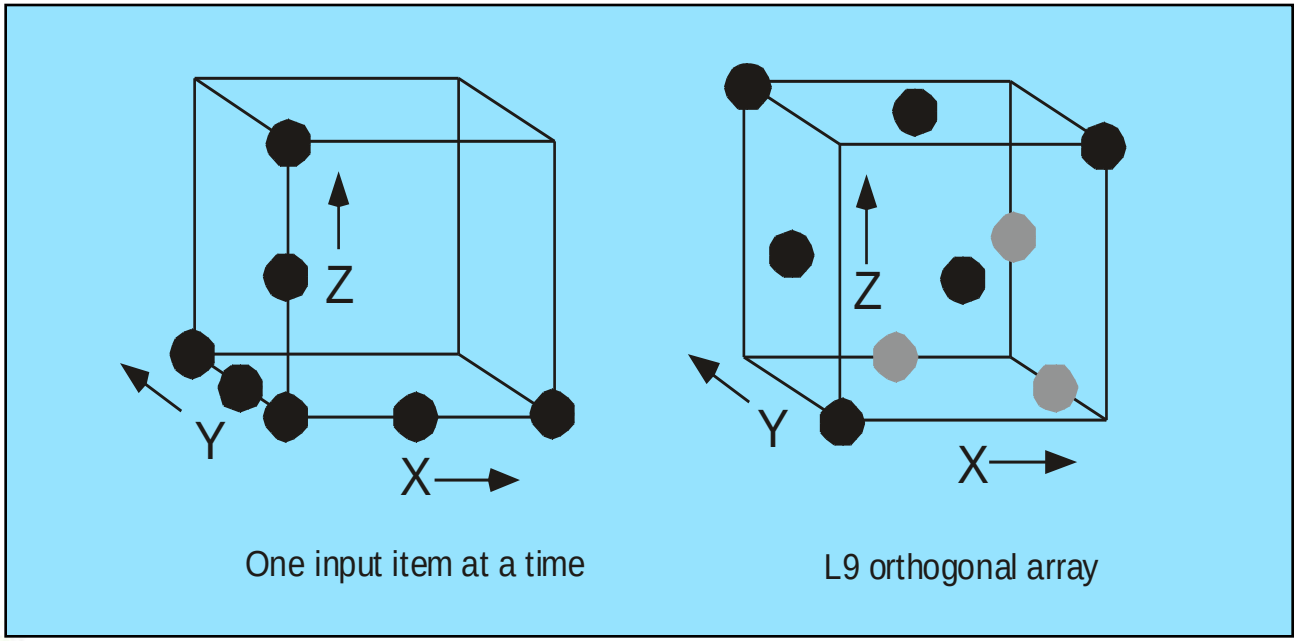
- data outside bounds of the program
- physically impossible data
- proper value supplied in wrong place

- **Boundary Value Analysis**



- **Comparison Testing**
  - Used only in situations in which the reliability of software is absolutely critical (e.g., human-rated systems)
    - 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 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]