SOFTWARE ENGINEERING

CHAPTER-15
TESTING CONVENTIONAL APPLICATIONS

Software Engineering: A Practitioner's Approach, 7th edition

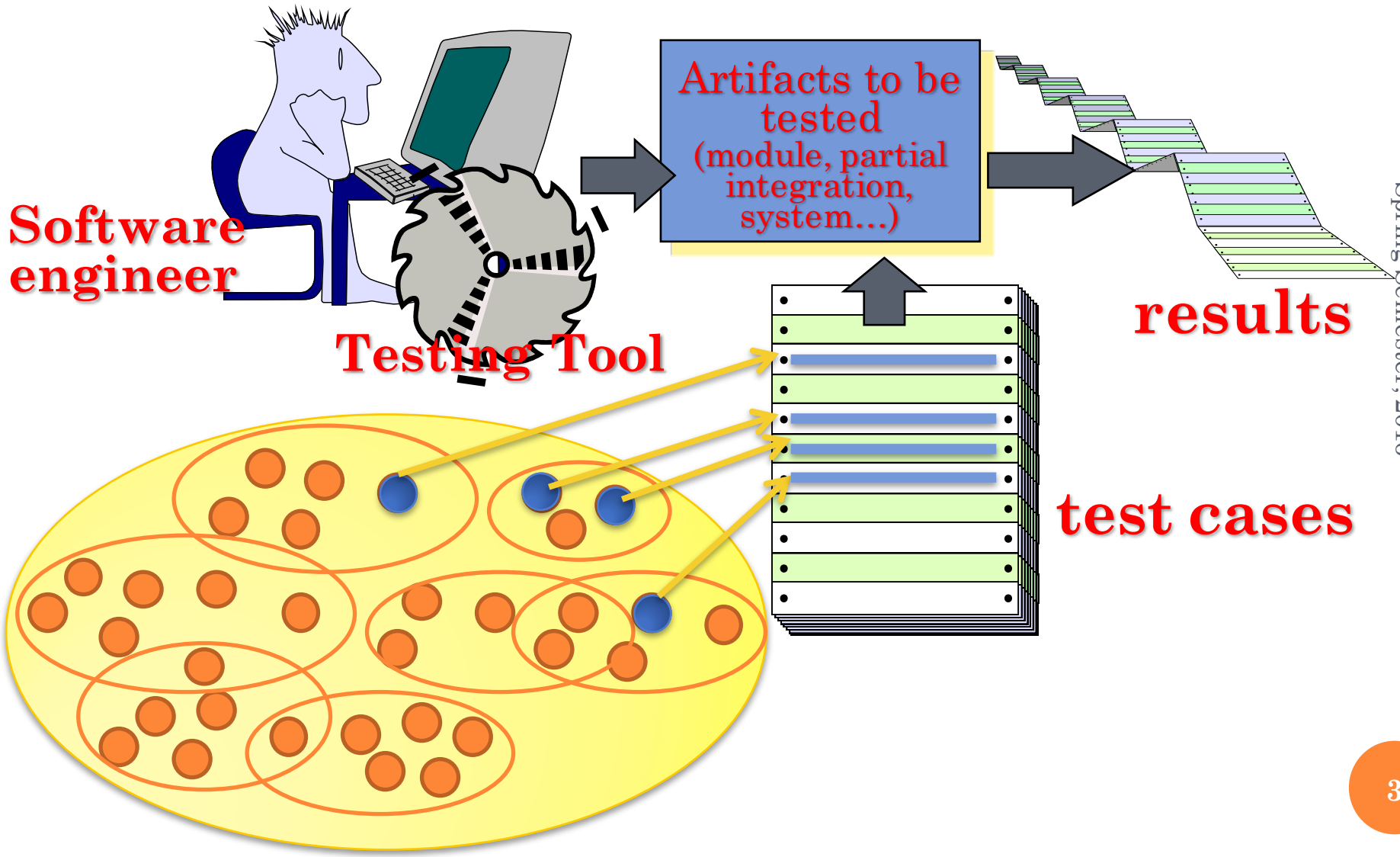
Originated by Roger S. Pressman

Software School, Fudan University
Spring Semester, 2016

Review: Testing Strategies

- Testing: finding errors prior to delivery to the end user
- Test Case: a set of conditions or variables to drive the tests
- Unit testing: module level, usually conducted by developers, driven by delivery
- Integration testing: to uncover errors associated with interfacing
  - Incremental integration: top-down, bottom-up
- High-order testing: Validation Testing, System Testing
- Debug: locate the problem's source from symptom

Testing: A sampling-based Process



Tradeoff between Cost and Completeness

Careful Tradeoff/Balance according to Quality Criteria and Time/Cost Constraints

Coverage, Completeness, Sufficiency

Cost, Time to Market ……

Testing Tactics

- Test design: to design a series of test cases that have a high likelihood of finding errors
- How? follow well-proved testing tactics: these techniques provide systematic guidance for designing tests that
  - exercise the internal logic and interfaces
  - exercise the input and output domains

To uncover errors in program function, behavior, and performance

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
  - E.g. consider abnormal input or operation situation
- A good test is not redundant
  - Redundant test means useless in revealing new errors
- A good test should be "best of breed"
- A good test should be neither too simple nor too complex

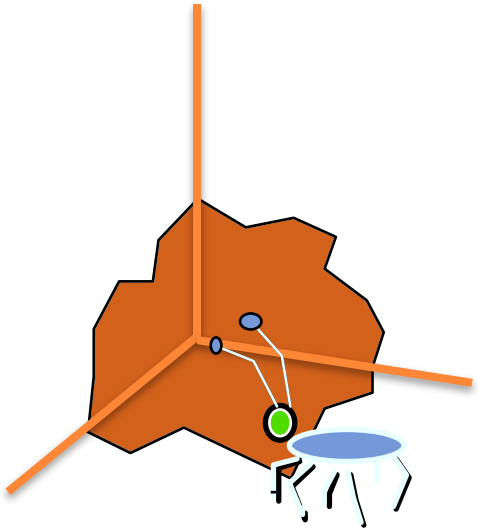
Internal and External Views

- 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 ..."

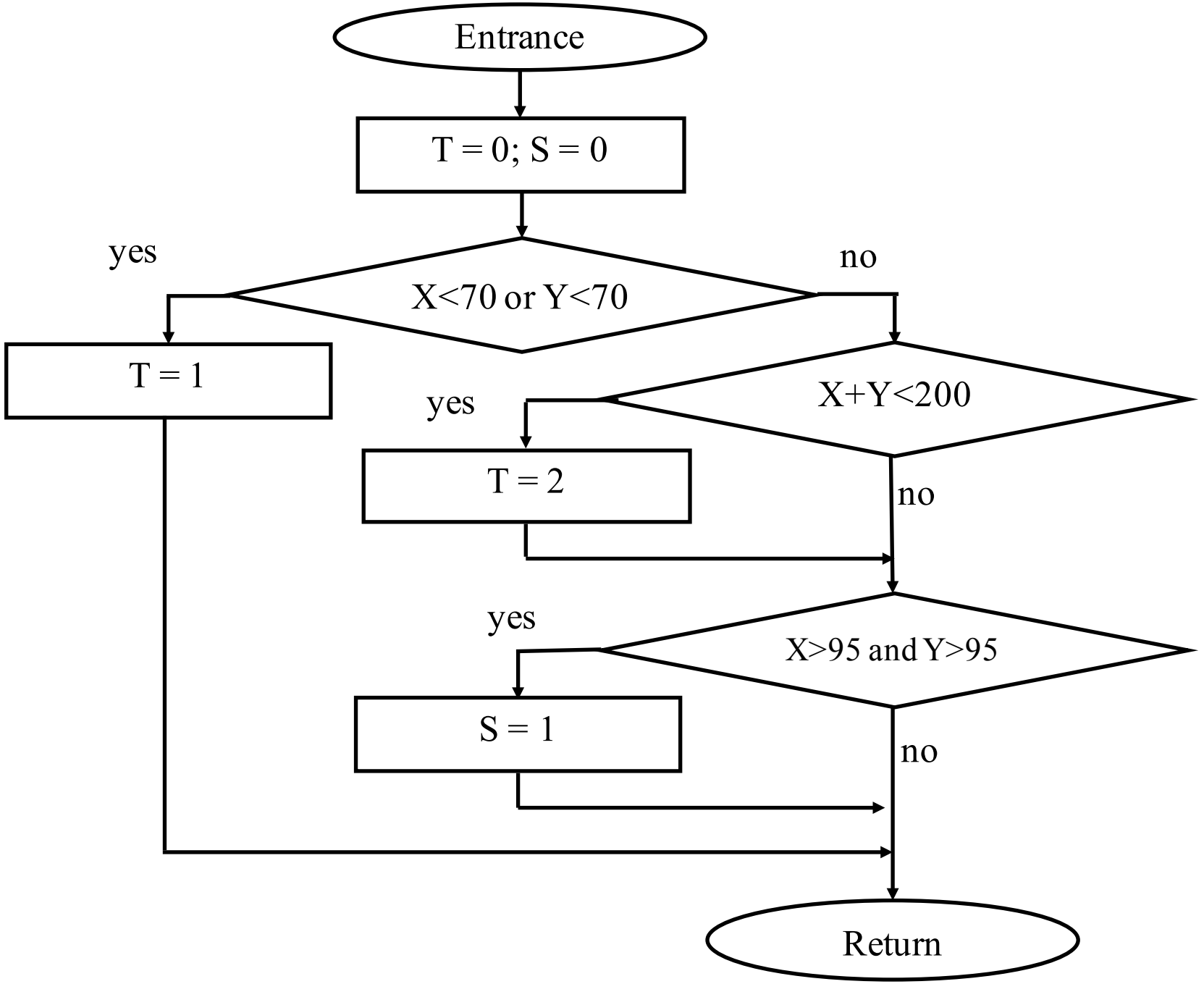
Boris Beizer



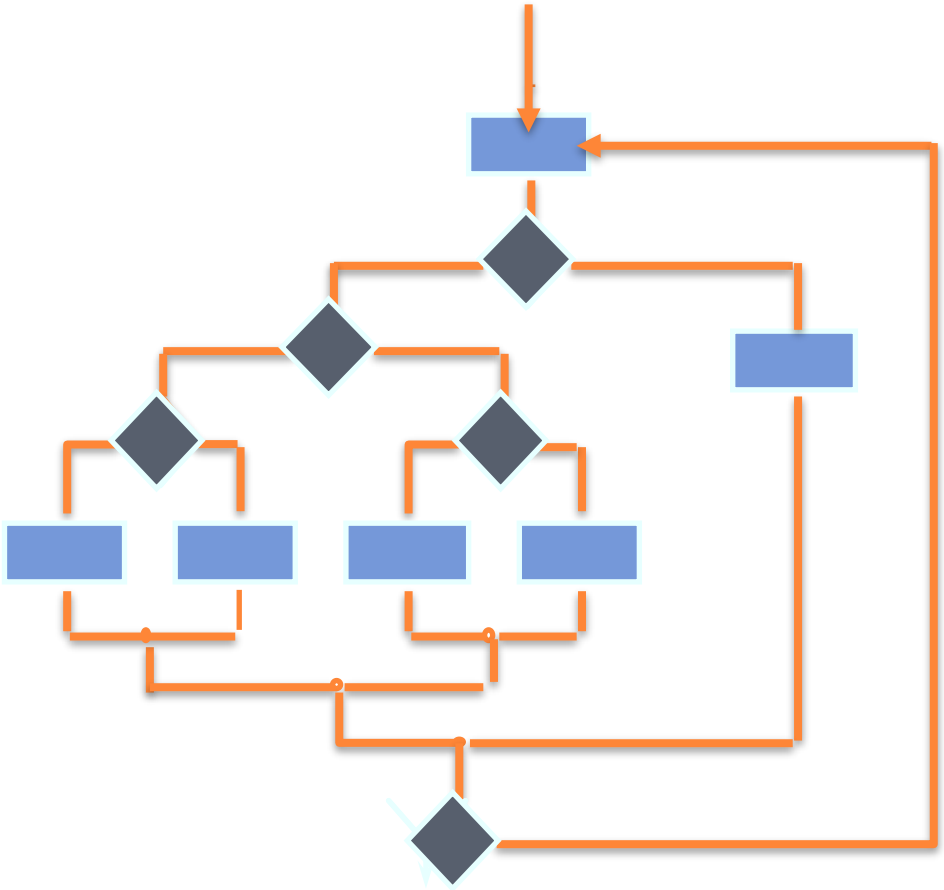
OBJECTIVE to uncover errors

CRITERIA in a complete manner

CONSTRAINT with a minimum of effort and time


Entrance
T = 0; S = 0
yes
X<70 or Y<70
no
T = 1
yes
X+Y<200
T = 2
no
yes
X>95 and Y>95
S = 1
no
Return

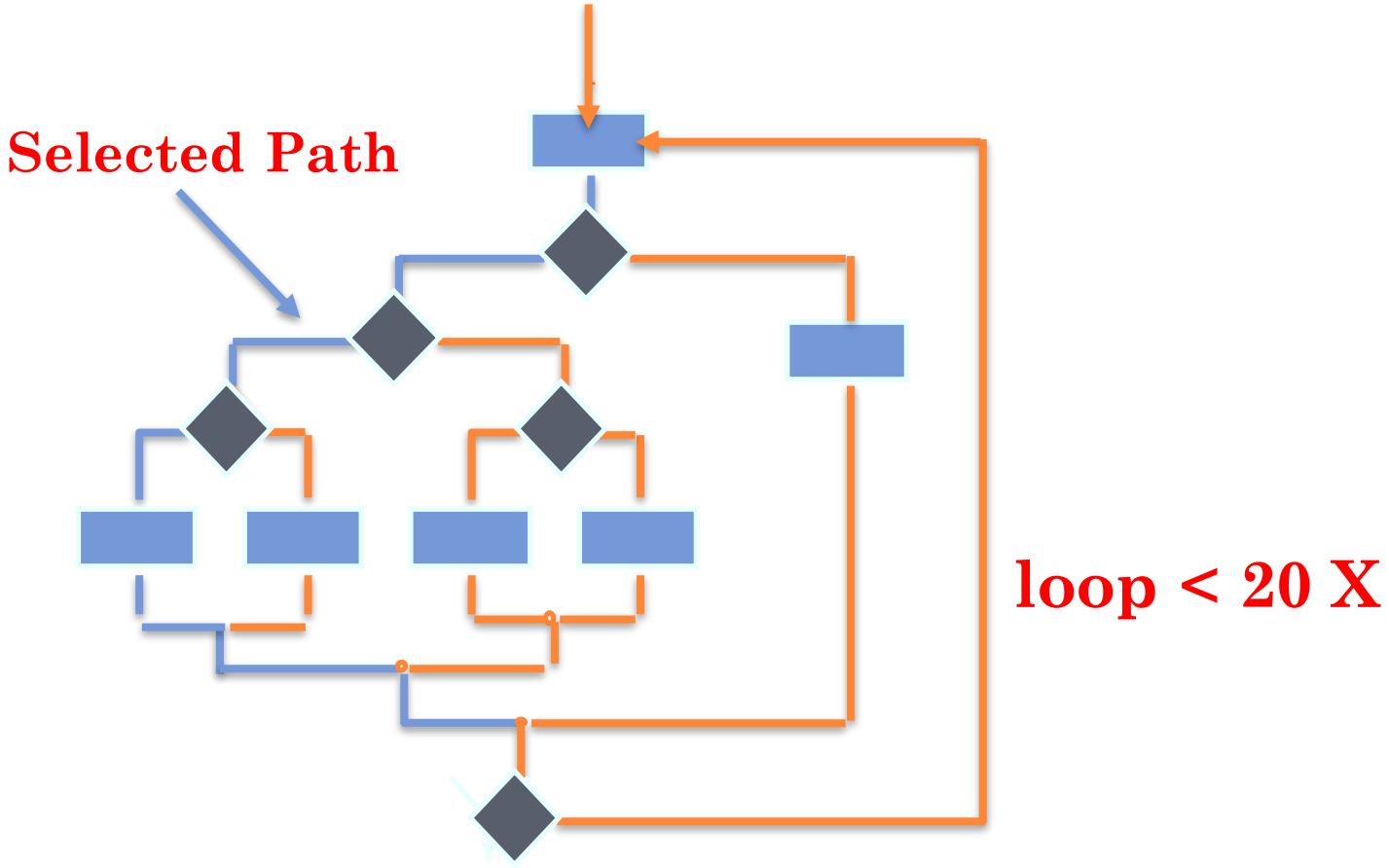
EXHAUSTIVE TESTING



loop < 20 X

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



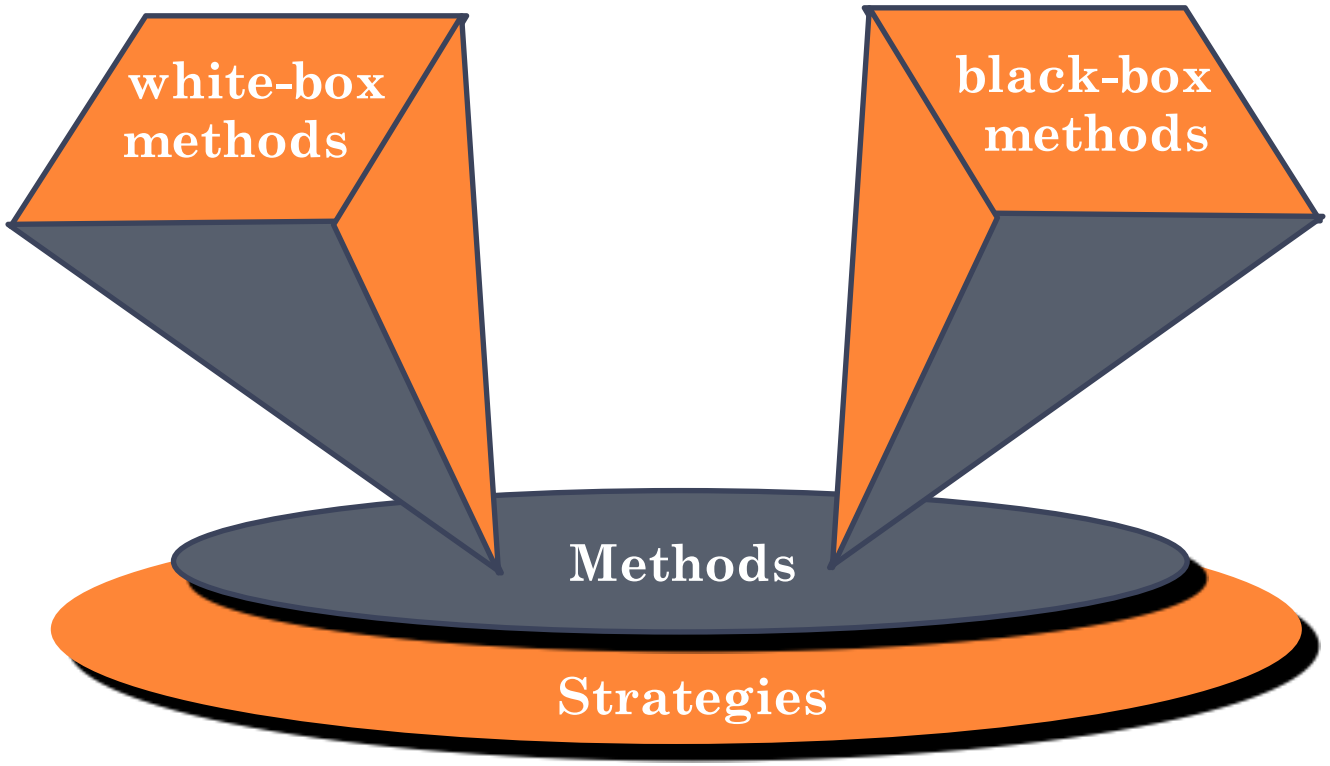
Testing Methods

Knowing the internal workings of a product

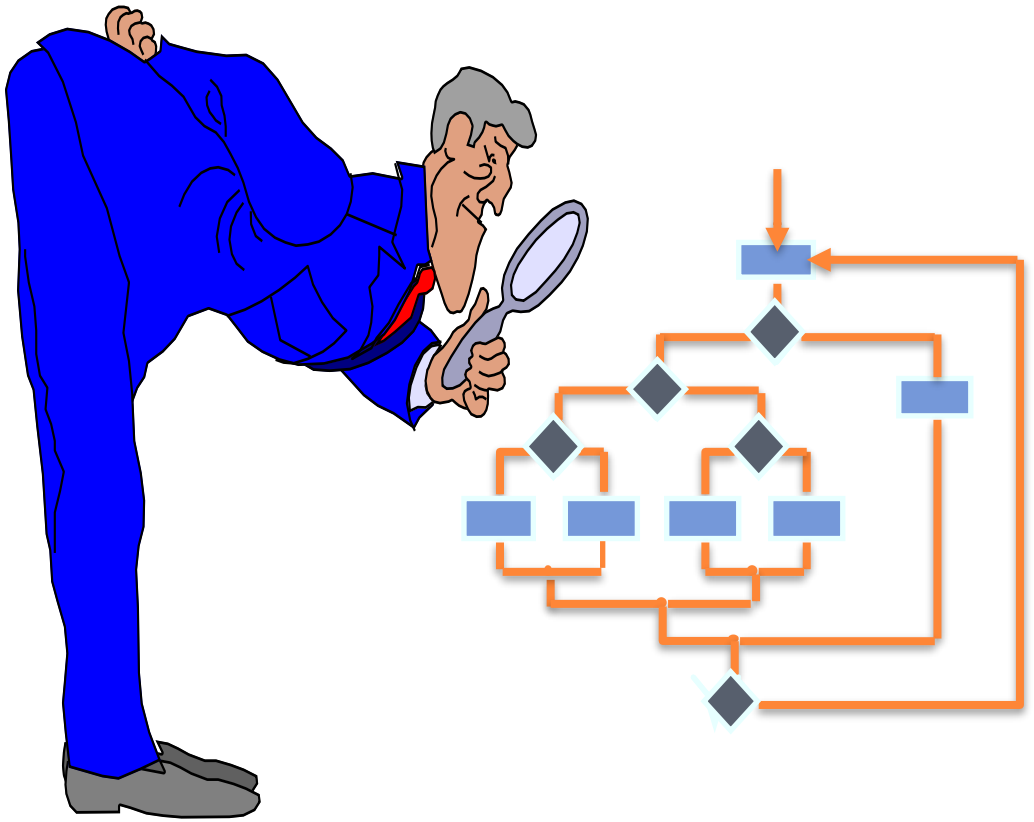
Tests are conducted to ensure that "all gears mesh"

Knowing the specified function provided via interfaces (external knowledge)

Tests are conducted that demonstrate each function is fully operational while searching for errors



White-Box Testing



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

Why Cover?

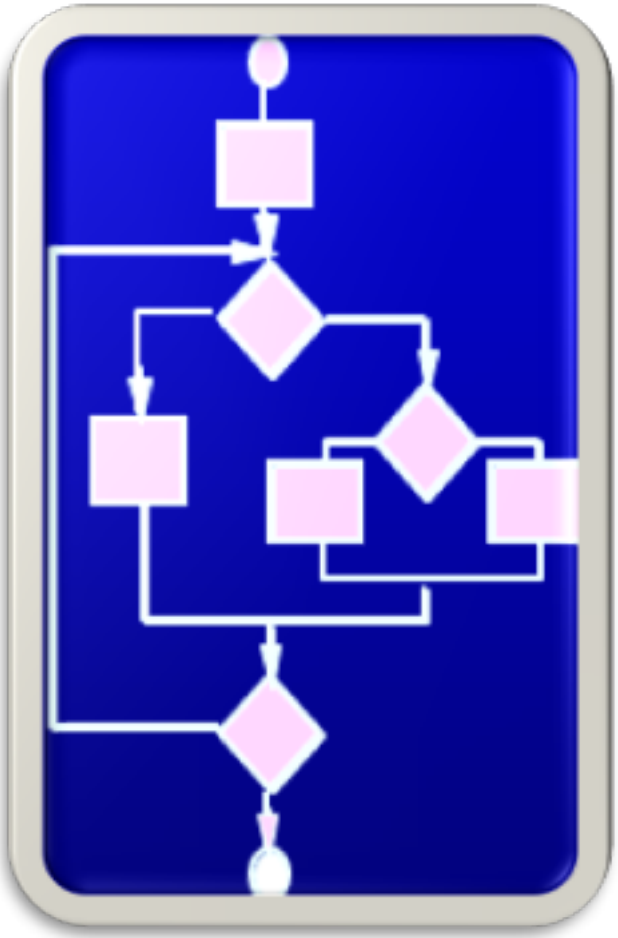
- Logic errors and incorrect assumptions are inversely proportional to a path's execution probability
- We often believe that a path is not likely to be executed; in fact, reality is often counter intuitive
- Typographical errors are random; it's likely that untested paths will contain some

White-Box Testing Criteria

- Guarantee that all independent paths are 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 structure to ensure their validity

White-Box Testing: Basis Path Testing

Derived test cases guarantee to execute every statement and conditional edge in the program at least one time during testing



First, we compute the cyclomatic complexity:

number of simple decisions+1

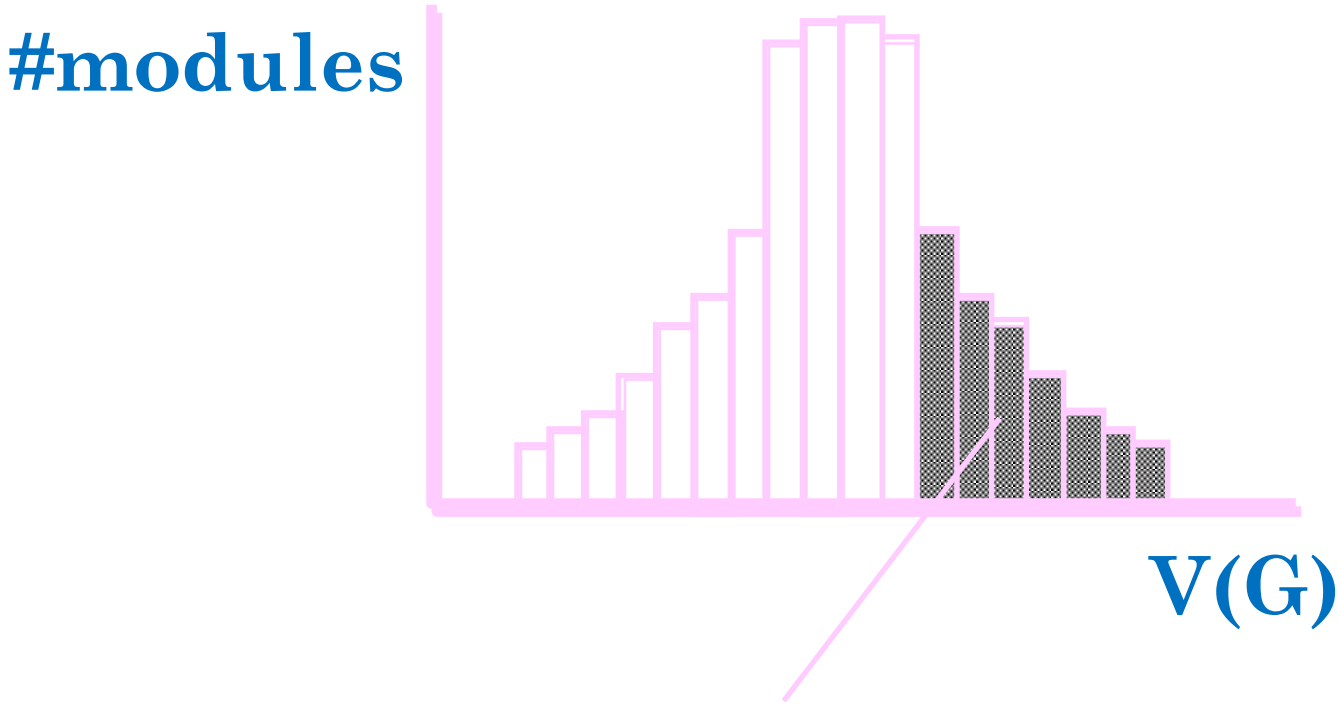
or

number of enclosed areas+1

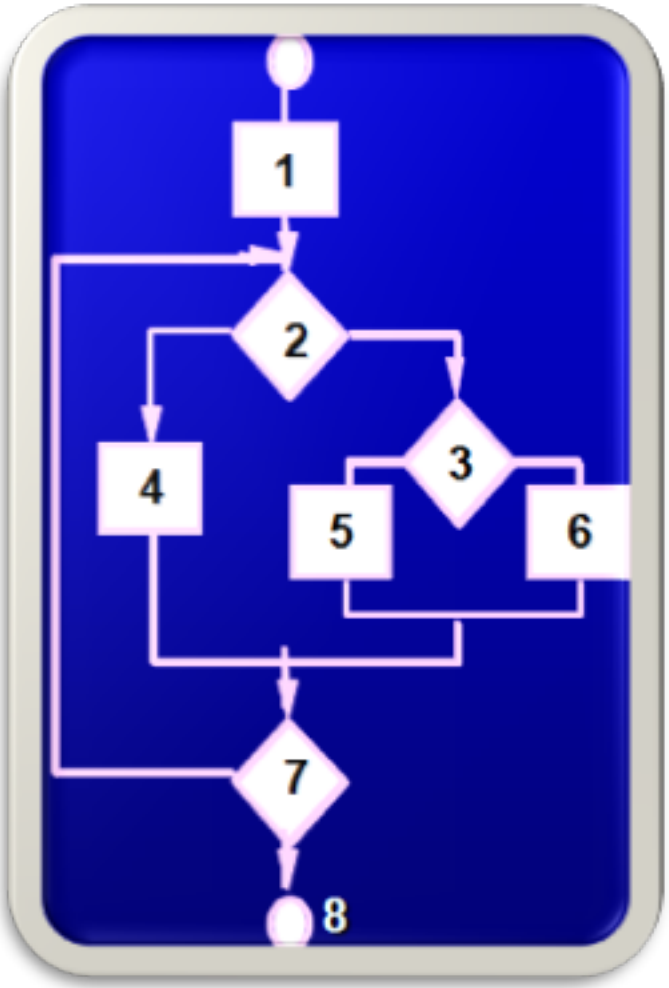
In this case, V(G) = 4

CYCLOMATIC COMPLEXITY

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



Basis Path Testing



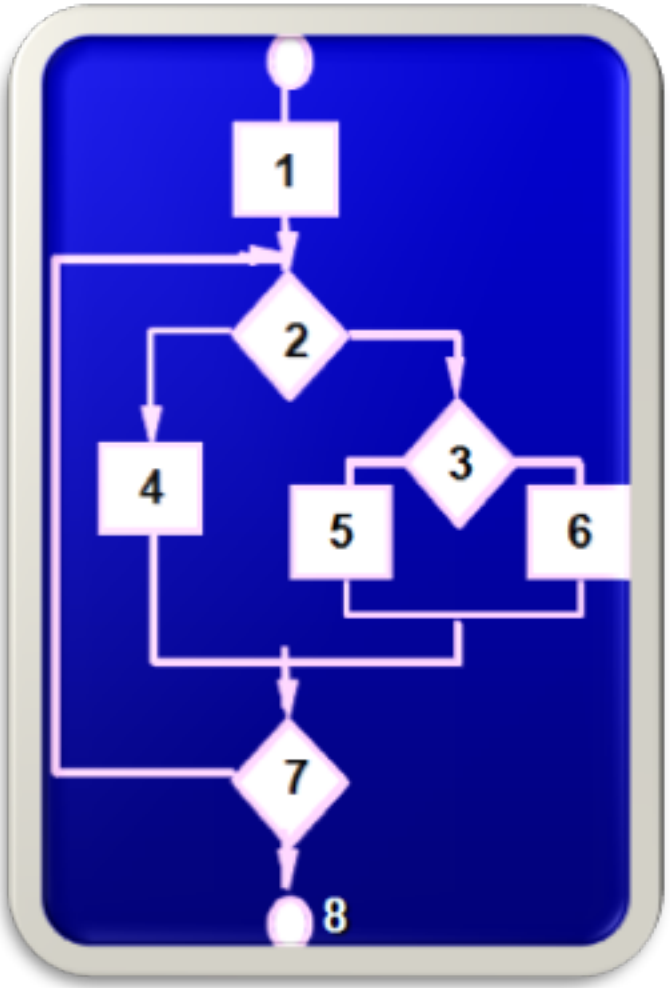
Next, we derive the independent paths:

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.

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

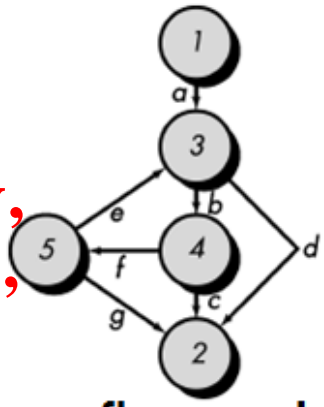
Deriving Test Cases

- *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.

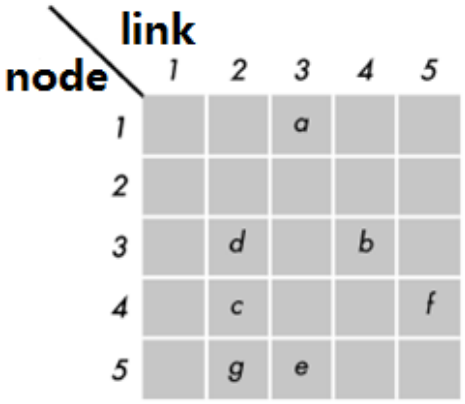
Graph Matrix: A Useful Data Structure

- Each row and column corresponds to an identified node
- 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

Link Weight: Connectivity, probability, processing time, memory, resource…



flow graph



matrix

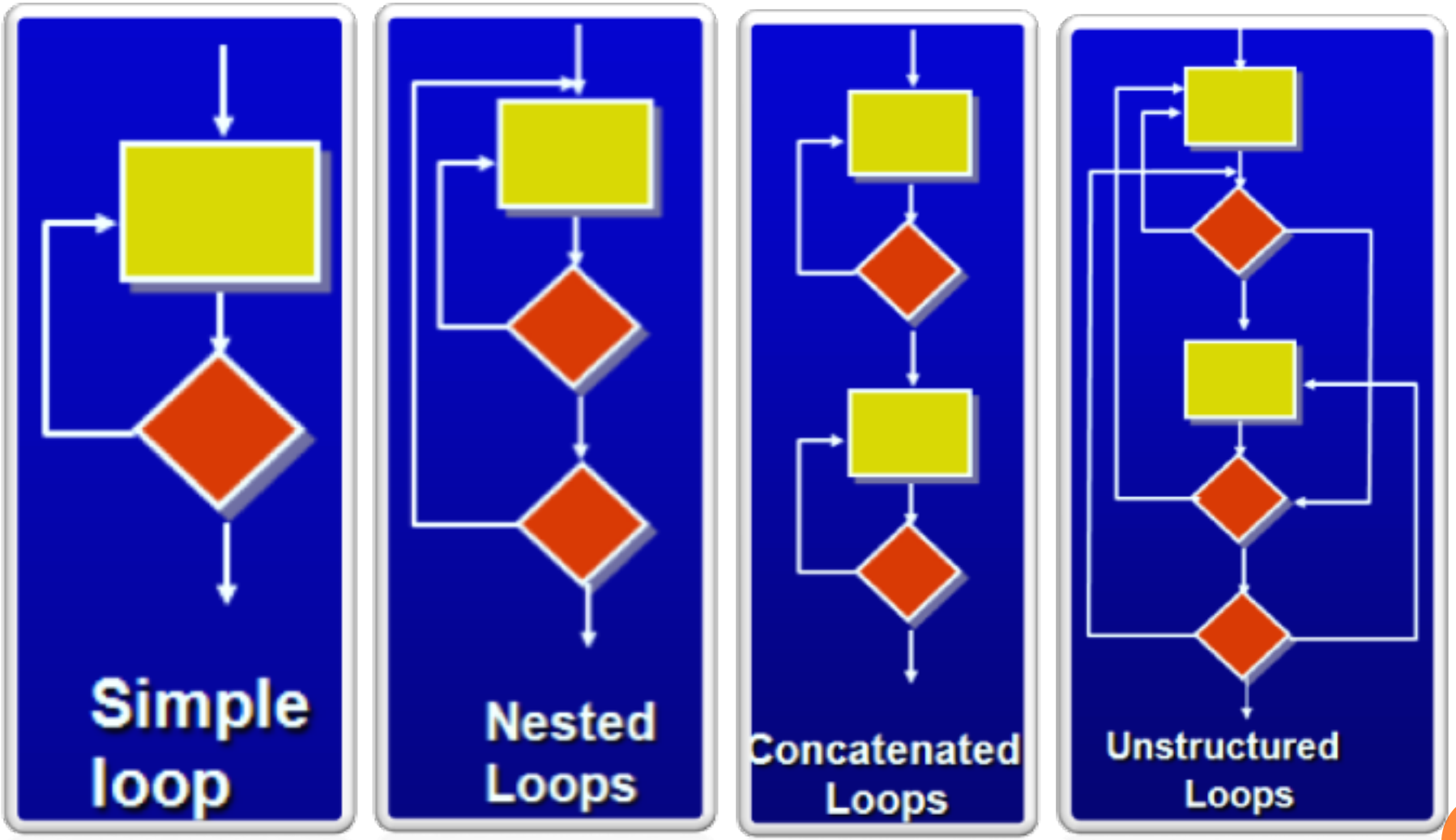
Control Structure Testing

- Basis Path Testing
- Condition testing — a test case design method that exercises the logical conditions contained in a program module (Conditions)
- Data flow testing — selects test paths of a program according to the locations of definitions and uses of variables in the program (Definition-Use Chains)
- Loop Testing

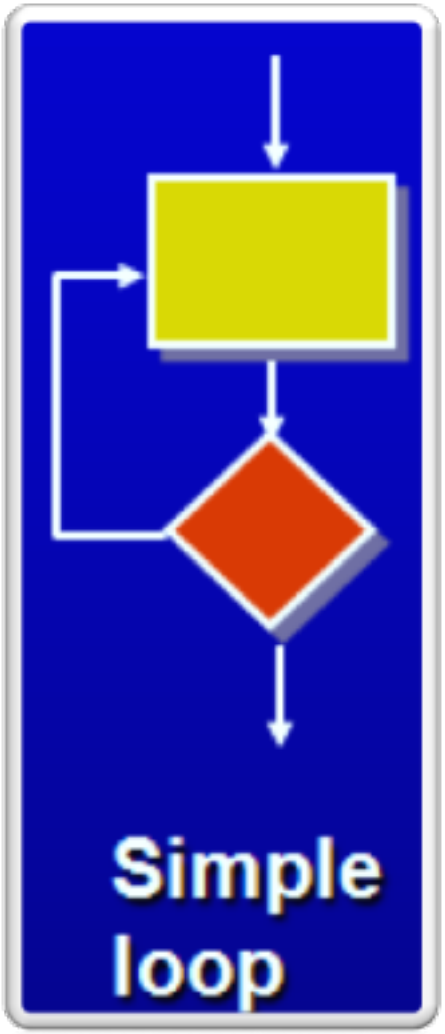
Data Flow Testing

- 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'*

Classes of Loop



Loop Testing: Simple Loops



Minimum conditions — Simple Loops

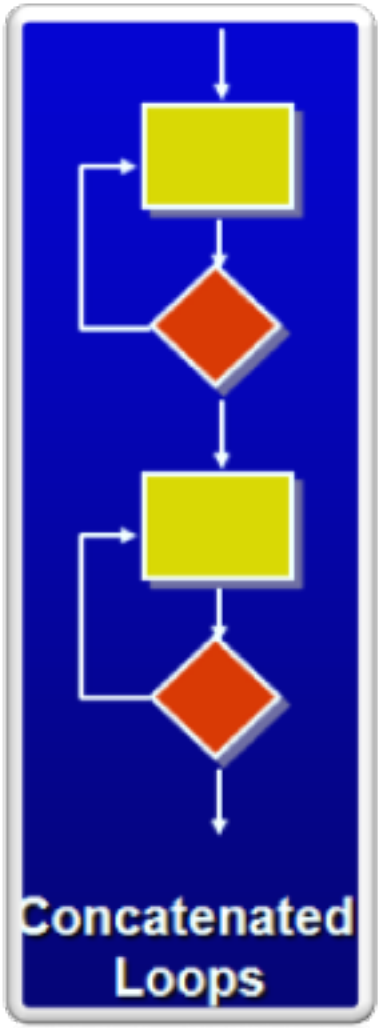
1. skip the loop entirely
2. only one pass through the loop
3. two passes through the loop
4. m passes through the loop m < n
5. (n-1), n, and (n+1) passes through the loop

where n is the maximum number of allowable passes

Loop Testing: Nested Loops

- 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.

Loop Testing: Concatenated Loops



If the loops are independent of one another

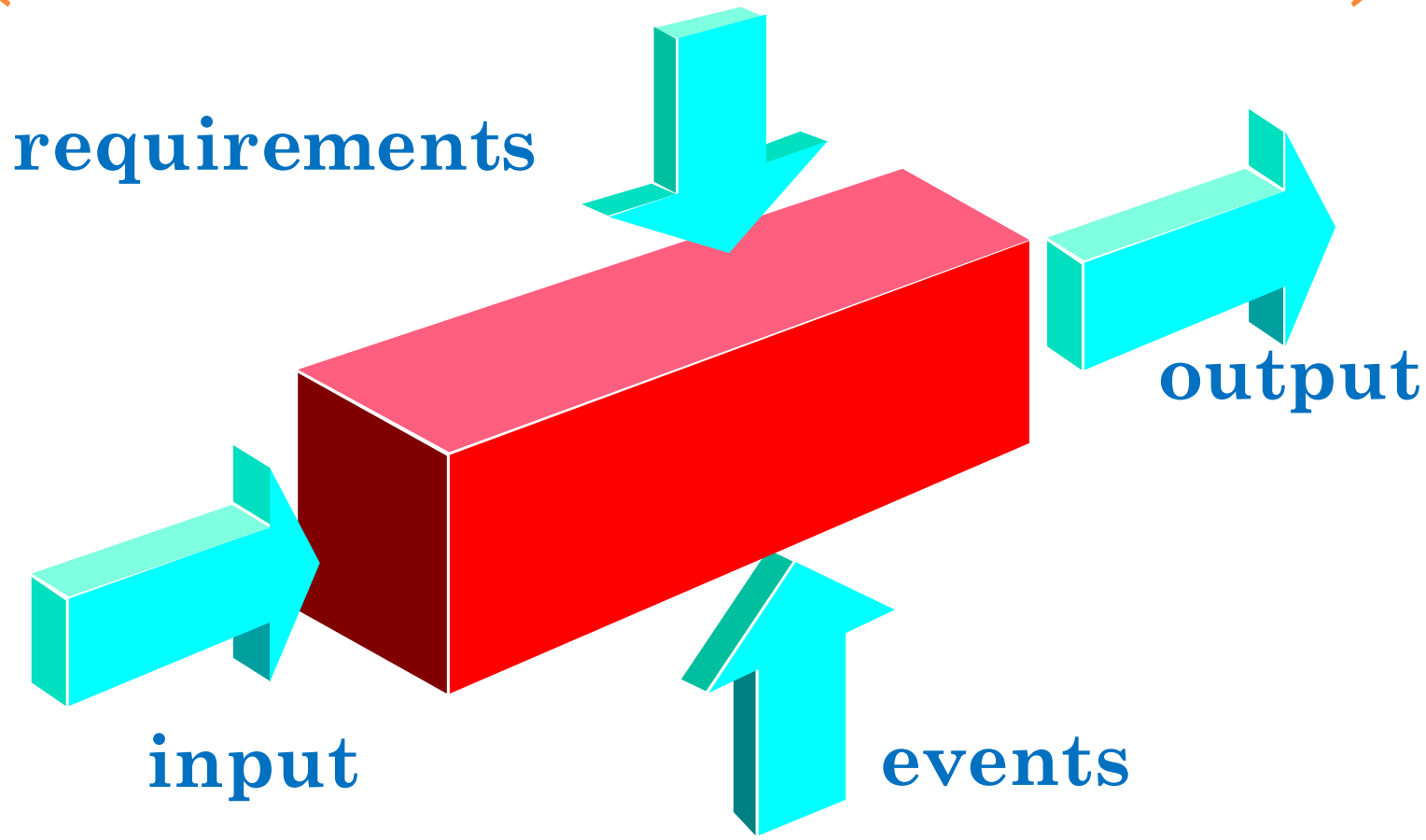
then treat each as a simple loop

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 (Behavior Testing)



- Focus on interfaces and information domain
- Tend to be applied during later stages of testing

Black-Box Testing

- Attempt to find the following errors
  - Incorrect or missing functions
  - Interface errors
  - Errors in data structures or external data base access
  - Behavior or performance error
  - Initialization and termination errors

Black-Box Testing: Questions to Be Answered

- 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?

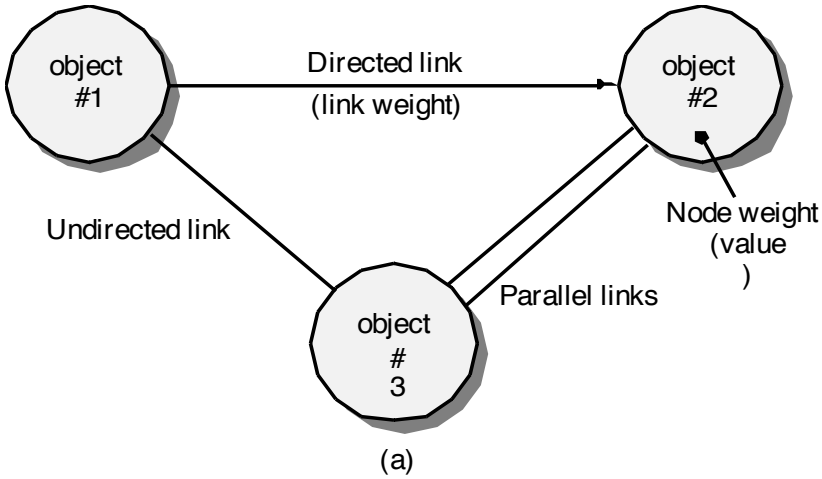
Black-Box Testing Methods

- Graph-based Methods
- Equivalence Partitioning
- Boundary Value Analysis
- Orthogonal Array Testing

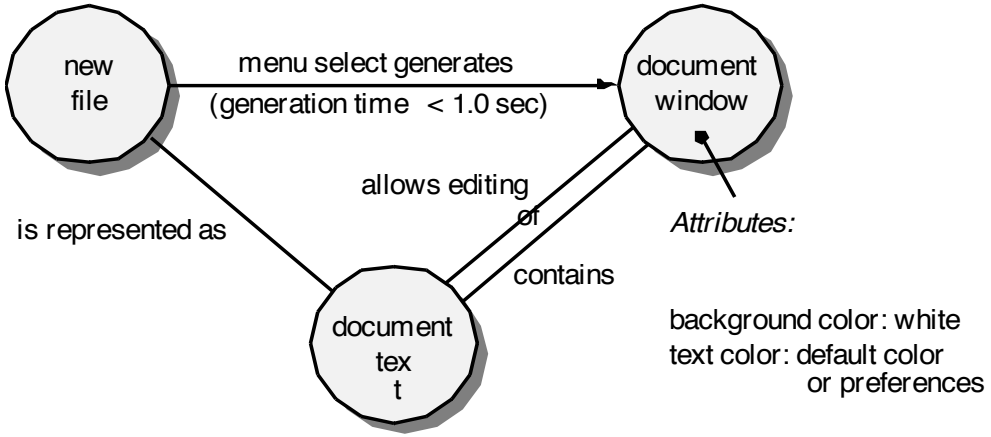
GRAPH-BASED METHODS

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)

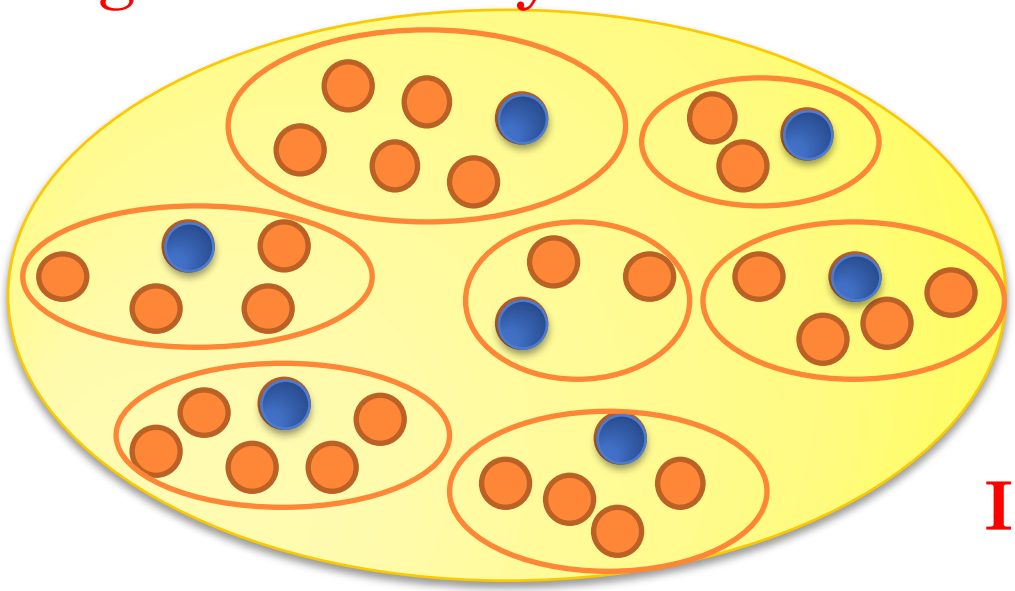


(b)

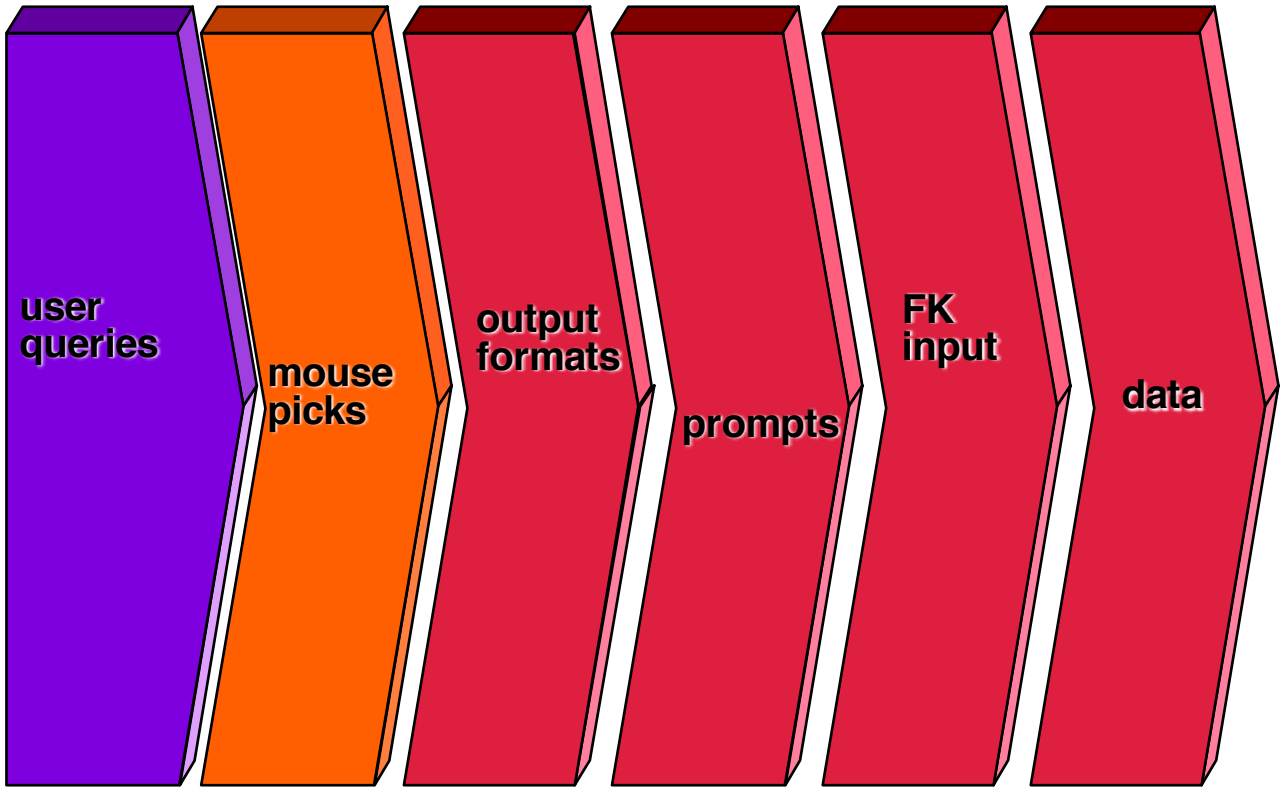
Test cases can be derived by traversing the graph and covering each of the relationships shown.

Equivalence Partitioning

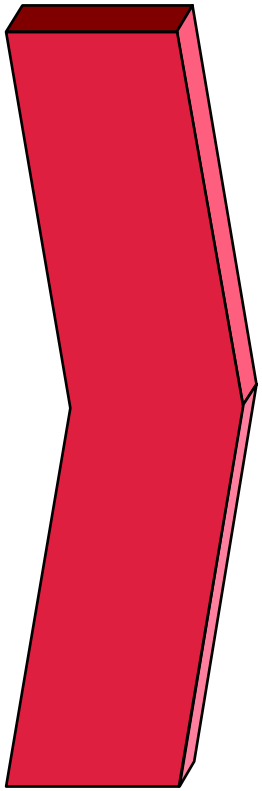
- Divide the input domain of a program into classes of data
- Then derive test cases from those classes of data
- An ideal test case can uncover a class of errors single-handedly



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 Class Definition Guidelines

Situation	Guideline
An input condition specifies a range	One valid and two invalid equivalence classes
An input condition requires a specific value	One valid and two invalid equivalence classes
An input condition specifies a member of a set	One valid and one invalid equivalence classes
If an input condition is boolean	One valid and one invalid equivalence classes

An Example

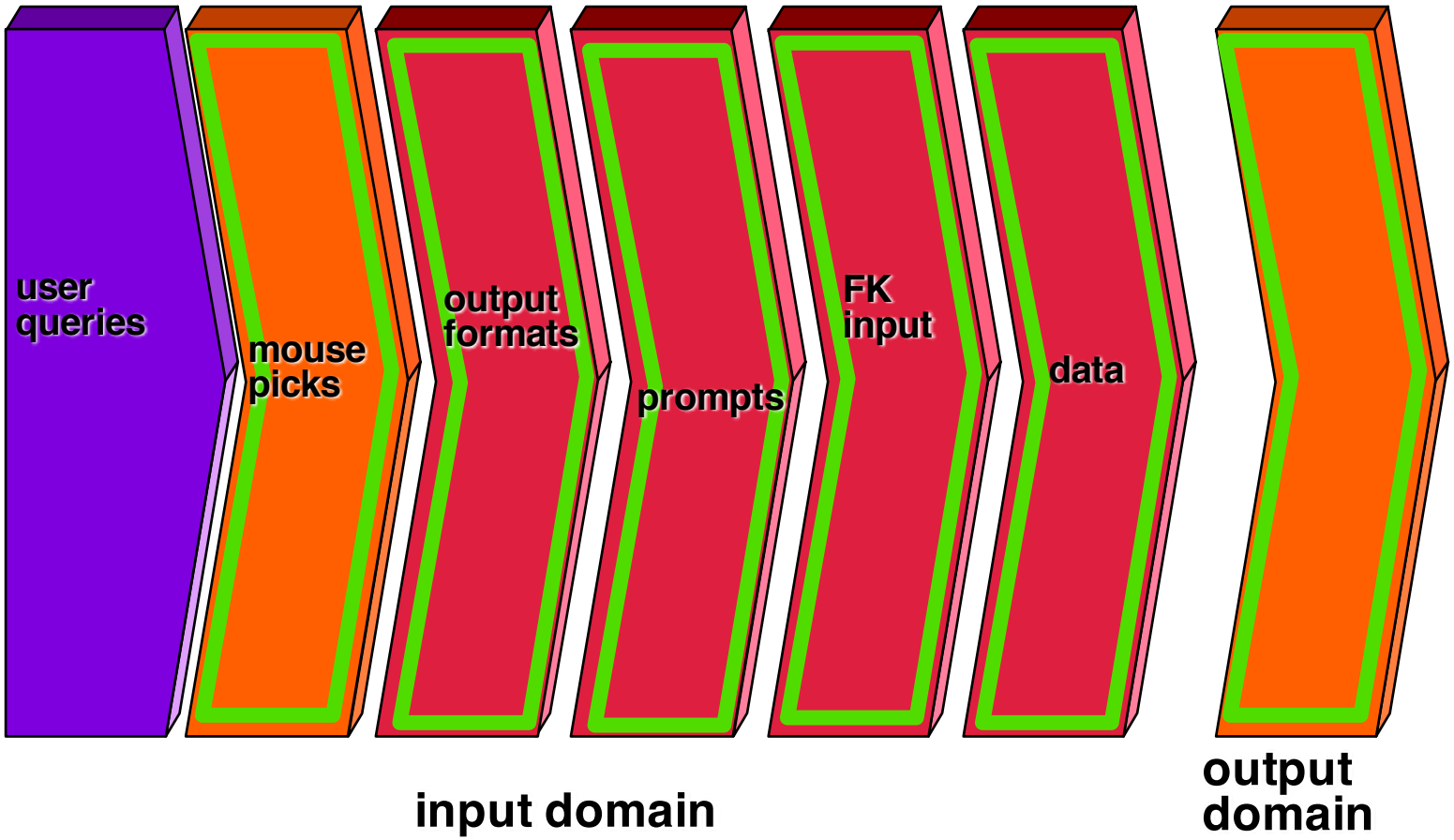
- To be tested: a method
  - Checks whether three lines with the given lengths can make a triangle
  - It takes three integers *a*, *b* and *c* as parameters, and returns TRUE (for yes) or FALSE (for no)
- Test cases

Condition	a	b	c	returns
Valid condition	3	4	5	TRUE
a+b<=c	1	1	2	FALSE
a<0	-1	4	5	FALSE
……				

Boundary Value Analysis (BVA)

- A greater number of errors occur at the boundaries of the input domain
- BVA complements equivalence partitioning: lead to selection of test cases at the "edges" of the class
- BVA derives test cases from the output domain as well

Boundary Value Analysis



BVA Guidelines

Situation	Guideline
An input condition specifies a range bounded by values *a* and *b*	*a* and *b* just above and just below *a* and *b*
An input condition specifies a number of values	minimum and maximum numbers just above and just below minimum and maximum
Apply the above 2 guidelines to output conditions	
If an internal data structure has prescribed boundaries	exercise the data structure at its boundary

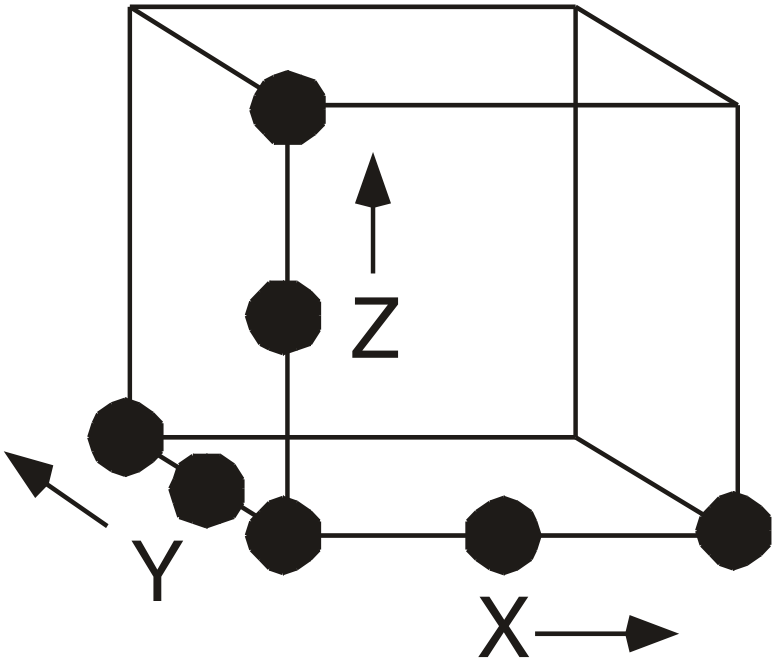
Comparison Testing

- 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

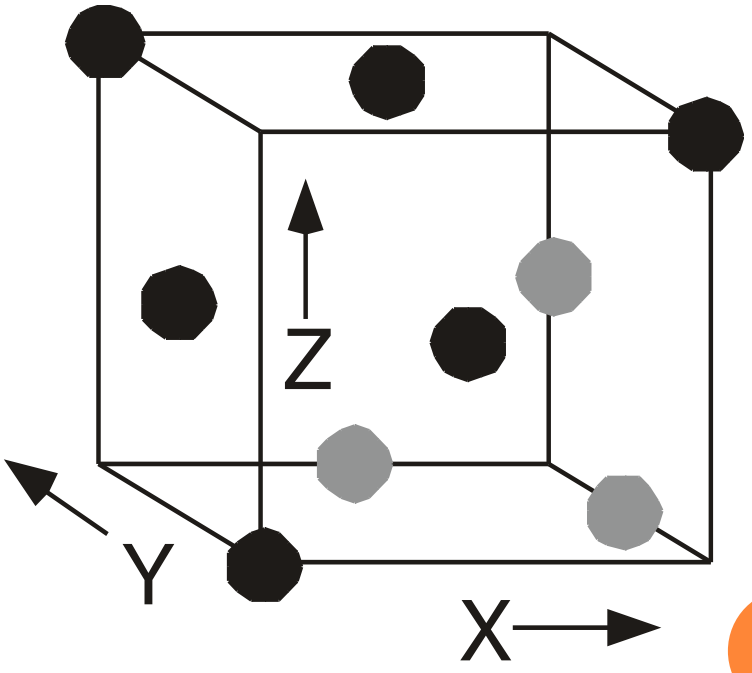
N-Version Programming

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



One input item at a time



L9 orthogonal array

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]

End of Chapter 18