Chapters 22, 23, 24

- **22-Software Testing Strategies**
- **23-Testing Conventional Applications**
- **24-Testing Object-Oriented Applications**

Slide Set to accompany
Software Engineering: A Practitioner's Approach, 7/e
by Roger S. Pressman

Edited by S.-H. Mirian Sharif University of Technology- Computer Engineering Department - 2019

Chapter 22

- **Software Testing Strategies**

Slide Set to accompany

Software Engineering: A Practitioner's Approach, 8/e

by Roger S. Pressman and Bruce R. Maxim

Software Testing

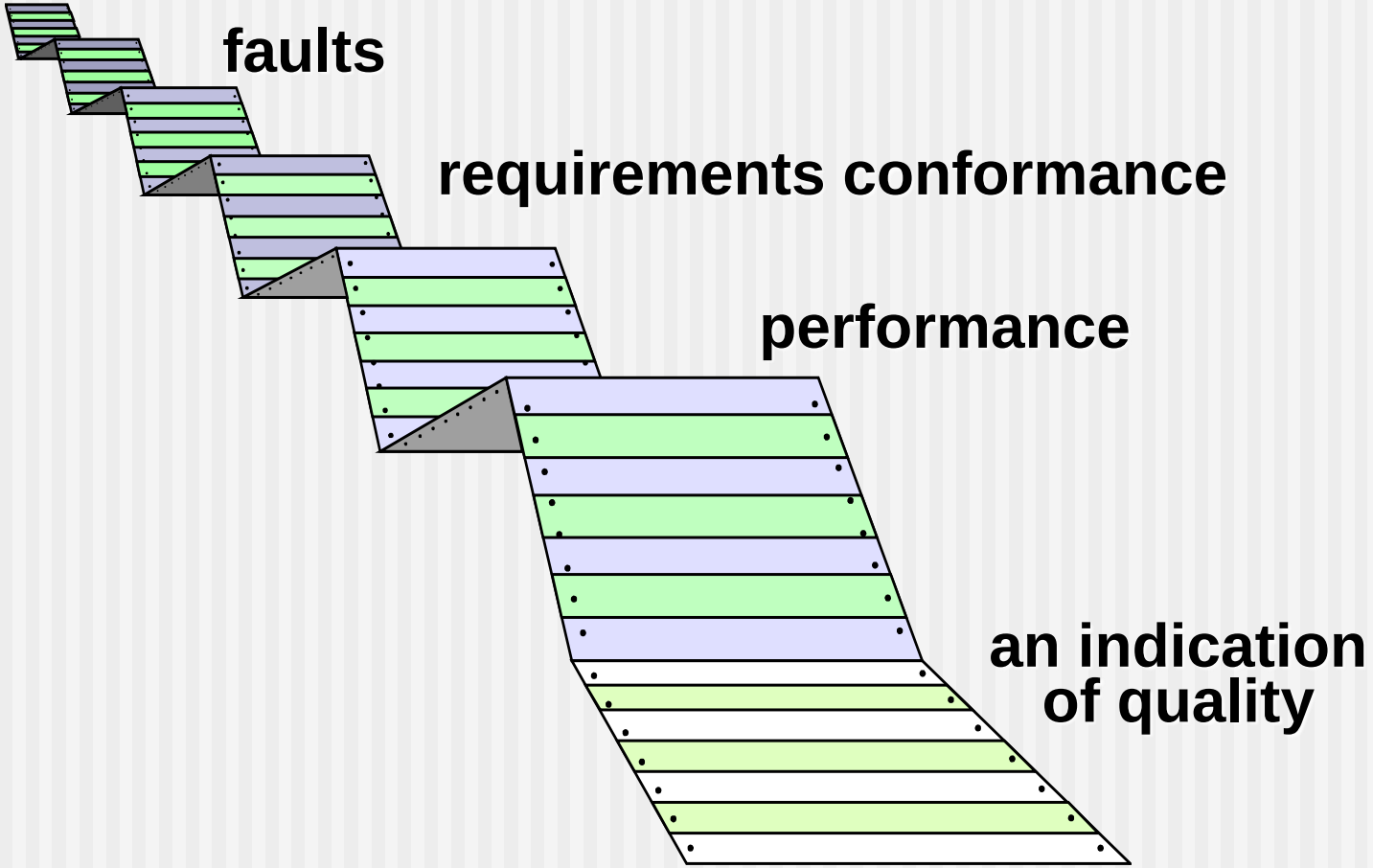
Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

Software Faults, Errors & Failures

- Software Fault :
  - A static defect in the software
- Software Error :
  - An incorrect internal state that is the manifestation of some fault
- Software Failure :
  - External, incorrect behavior with respect to the requirements or other description of the expected behavior

Faults in software are design mistakes and will always exist

What Testing Shows



Strategic Approach

- To perform effective testing, you should conduct effective technical reviews. By doing this, many faults will be eliminated before testing commences.
- Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and (for large projects) an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

Verification & Validation(V & V)

- *Verification*
  - refers to the set of tasks that ensure that software correctly implements a specific function.
- *Validation*
  - refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.
- Boehm [Boe81] states this another way:
  - *Verification:* "Are we building the product right?"
  - *Validation:* "Are we building the right product?"

Who Tests the Software?

developer

Understands the system

but, will test "gently"

and, is driven by "delivery"

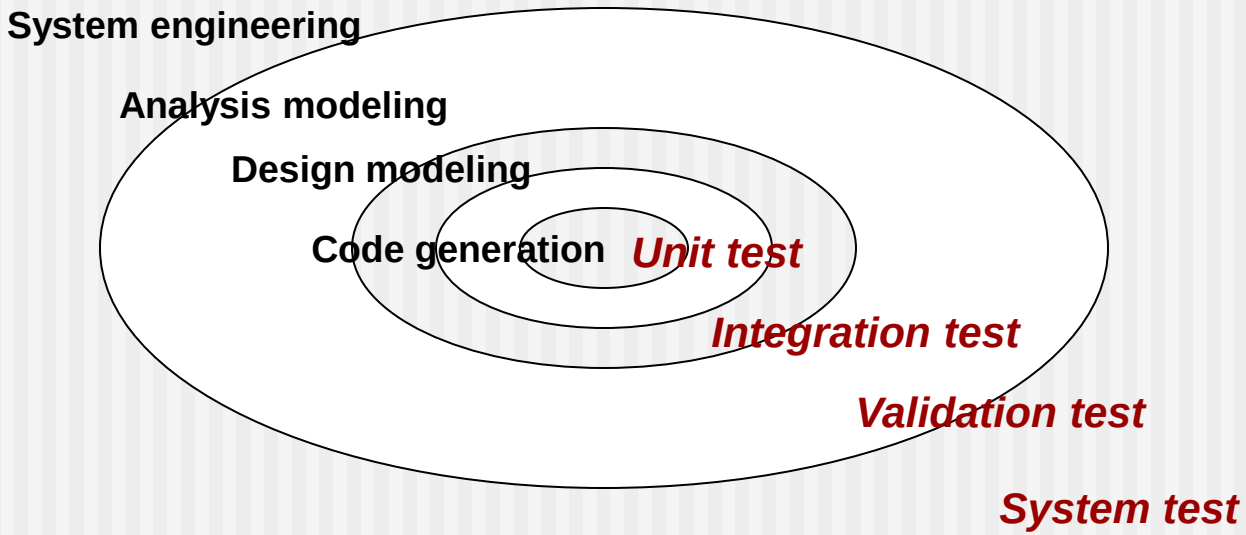
independent tester

Must learn about the system,

but, will attempt to break it

and, is driven by quality

Testing Strategy



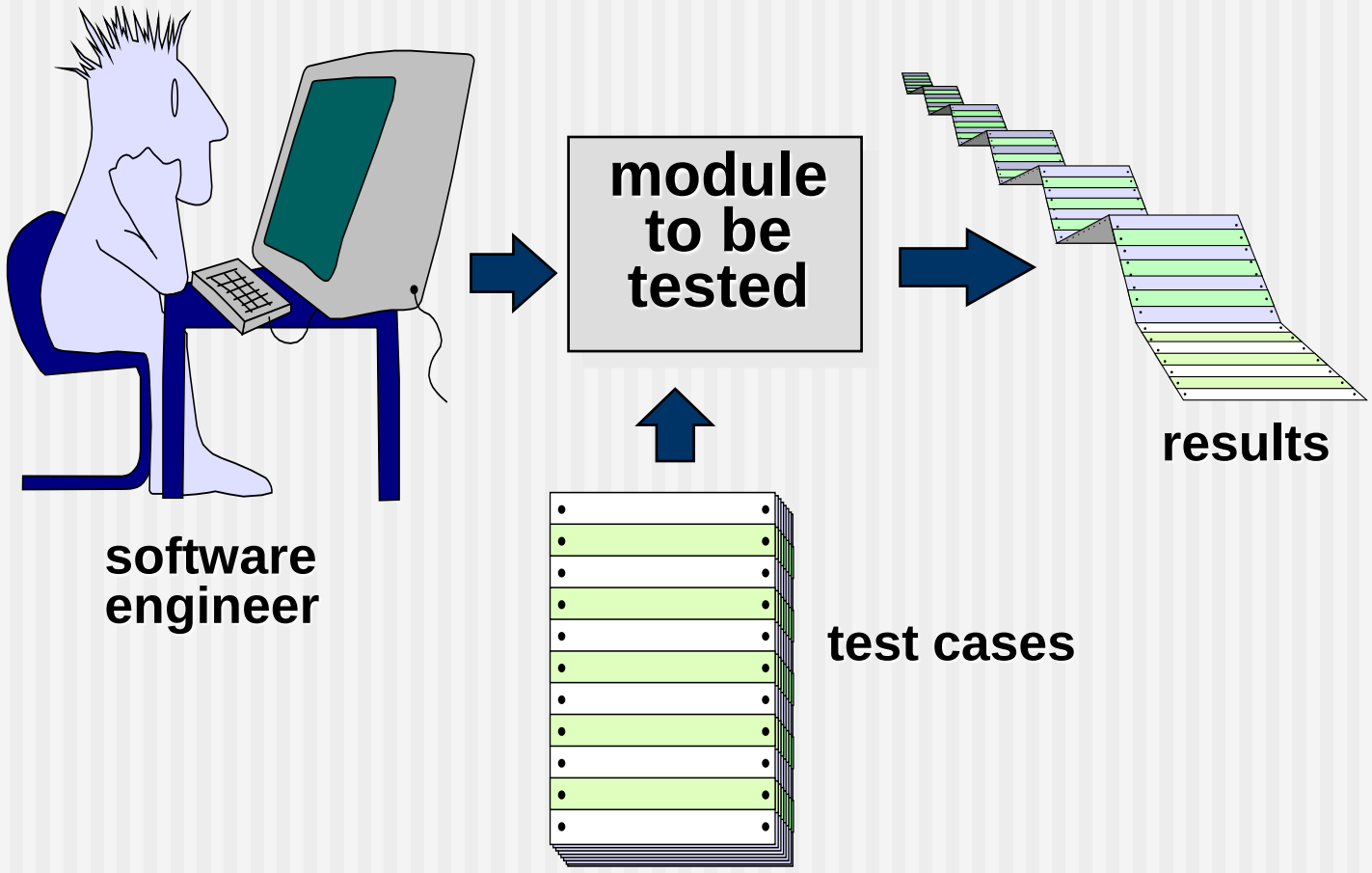
Testing Strategy

- We begin by ‘testing-in-the-small’ and move toward ‘testing-in-the-large’
- For conventional software
  - The module (component) is our initial focus
  - Integration of modules follows
- For OO software
  - our focus when “testing in the small” changes from an individual module (the conventional view) to an OO class that encompasses attributes and operations and implies communication and collaboration

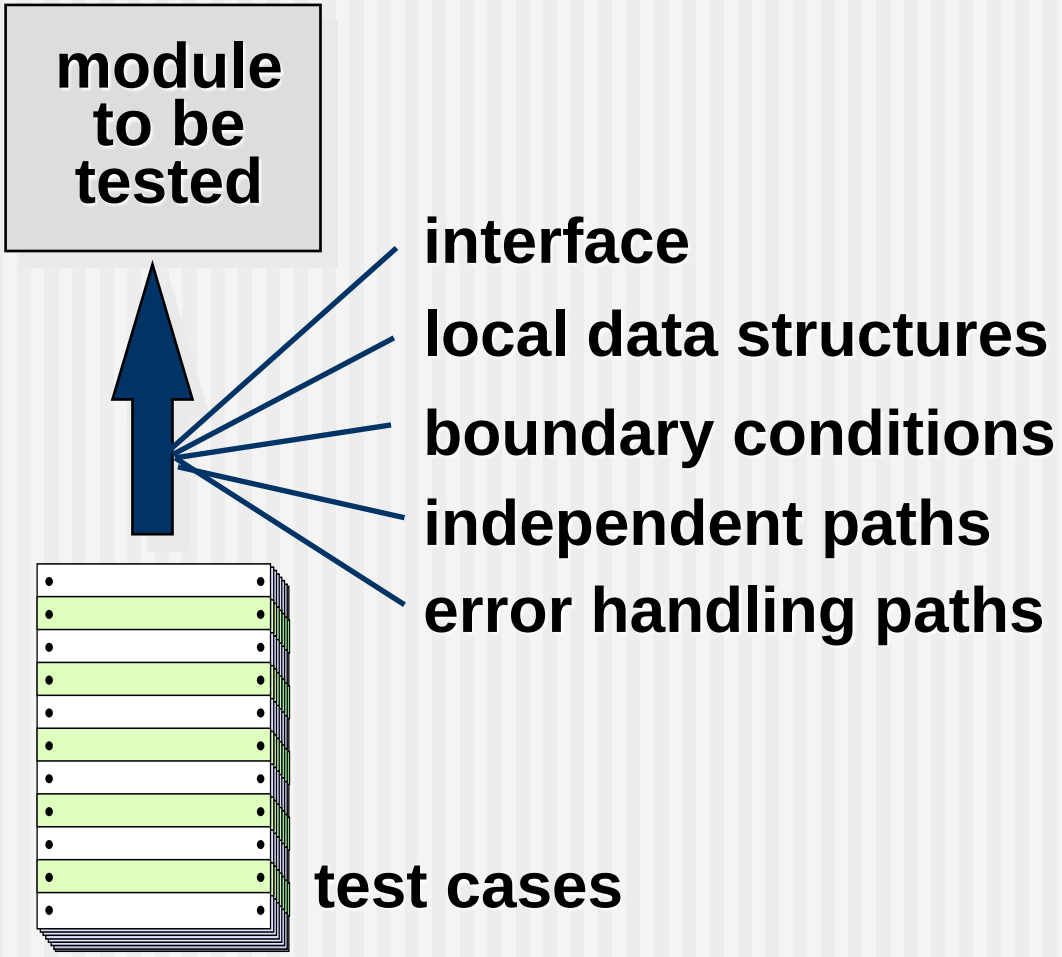
Strategic Issues

- Specify product requirements in a quantifiable manner long before testing commences.
- State testing objectives explicitly.
- Understand the users of the software and develop a profile for each user category.
- Develop a testing plan that emphasizes “rapid cycle testing.”
- Build “robust” software that is designed to test itself
- Use effective technical reviews as a filter prior to testing
- Conduct technical reviews to assess the test strategy and test cases themselves.
- Develop a continuous improvement approach for the testing process.

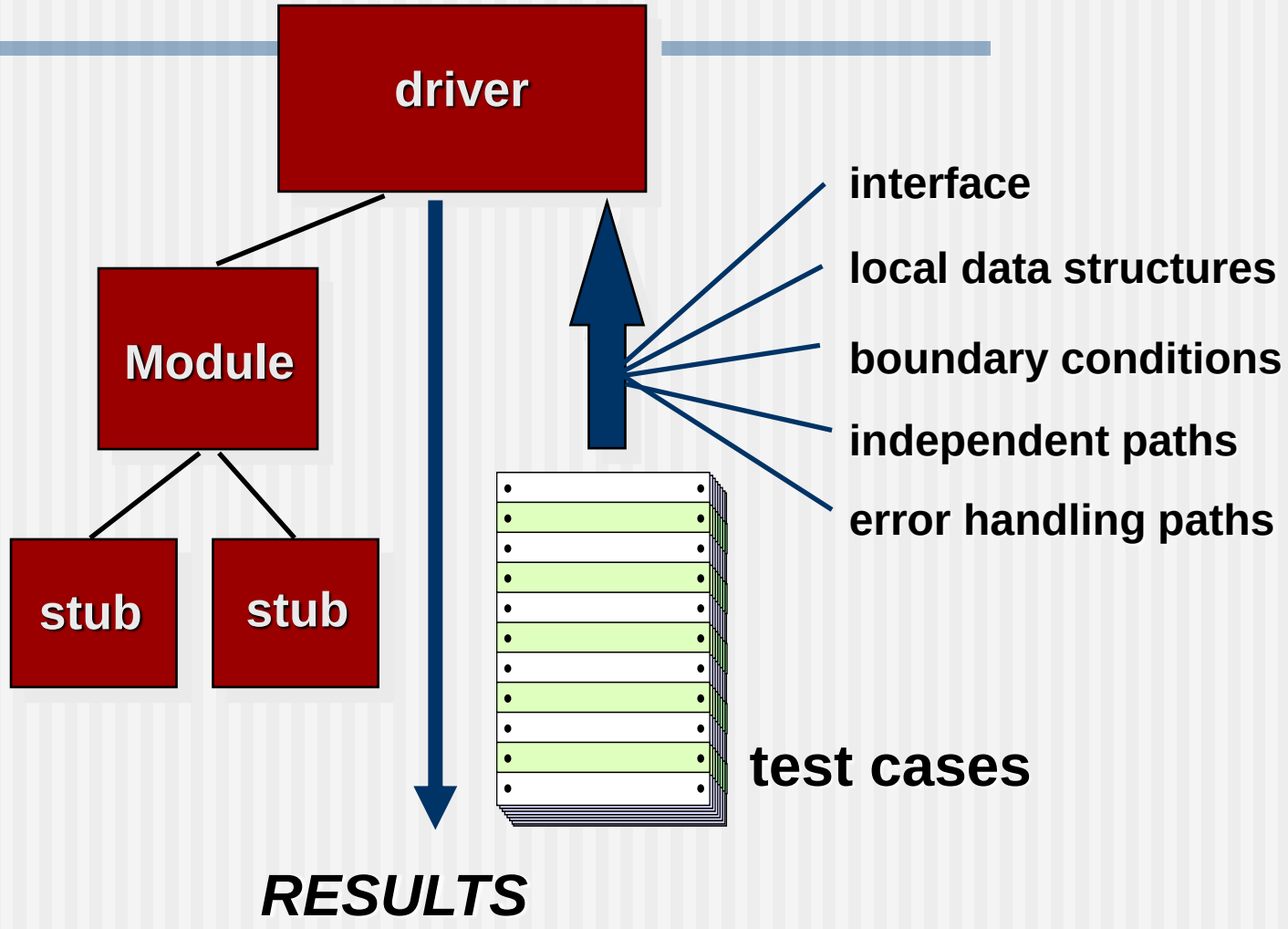
Unit Testing



Unit Testing



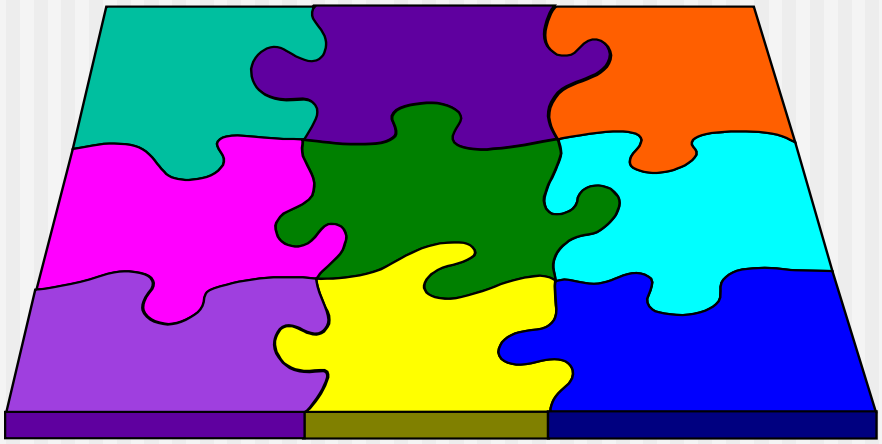
Unit Test Environment



Integration Testing Strategies

Options:

- **the “big bang” approach**
- **incremental construction strategies**
  - **top down**
  - **bottom up**



Regression Testing

- *Regression testing*
  - is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.
- Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.
- Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated capture/playback tools.

High Order Testing

- **Validation testing**
  - Focus is on software requirements
- **System testing**
  - Focus is on system integration
- **Alpha/Beta testing**
  - Focus is on customer usage
- **Recovery testing**
  - forces the software to fail in a variety of ways and verifies that recovery is properly performed
- **Security testing**
  - verifies that protection mechanisms built into a system will, in fact, protect it from improper penetration
- **Stress testing**
  - executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- **Performance Testing**
  - test the run-time performance of software within the context of an integrated system

Chapter 23

- **Testing Conventional Applications**

Slide Set to accompany

Software Engineering: A Practitioner's Approach, 8/e

by Roger S. Pressman and Bruce R. Maxim

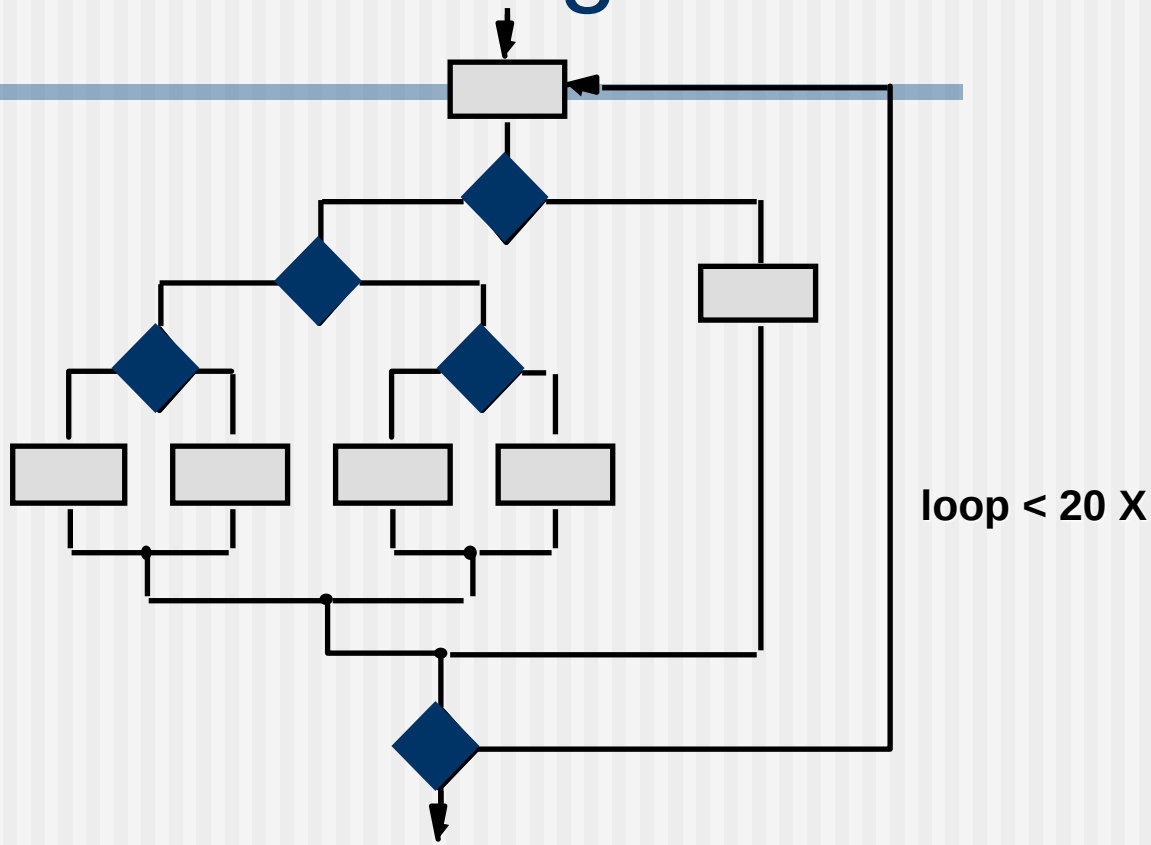
Testability

- 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?

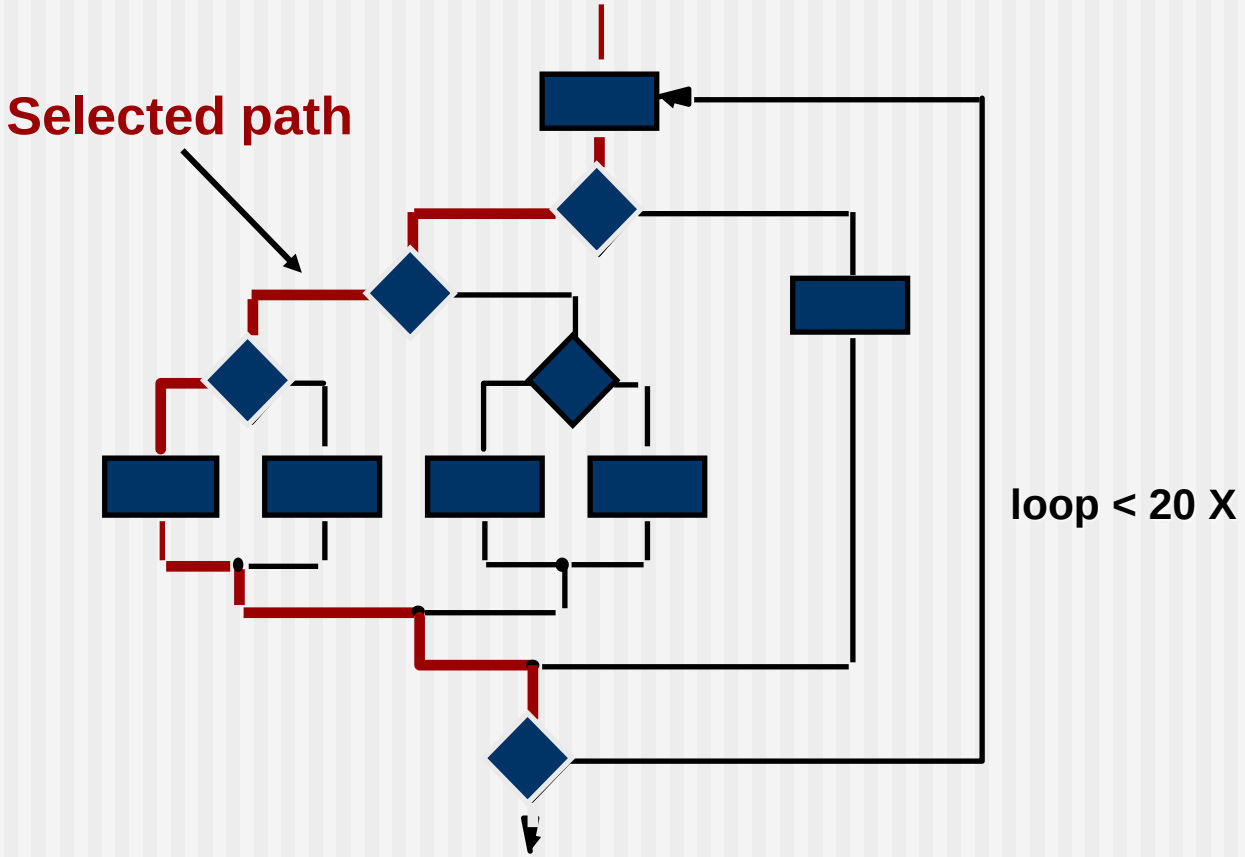
- A good test has a high probability of finding a fault.
- A good test is not redundant.
- A good test should be “best of breed”
- A good test should be neither too simple nor too complex

Exhaustive Testing

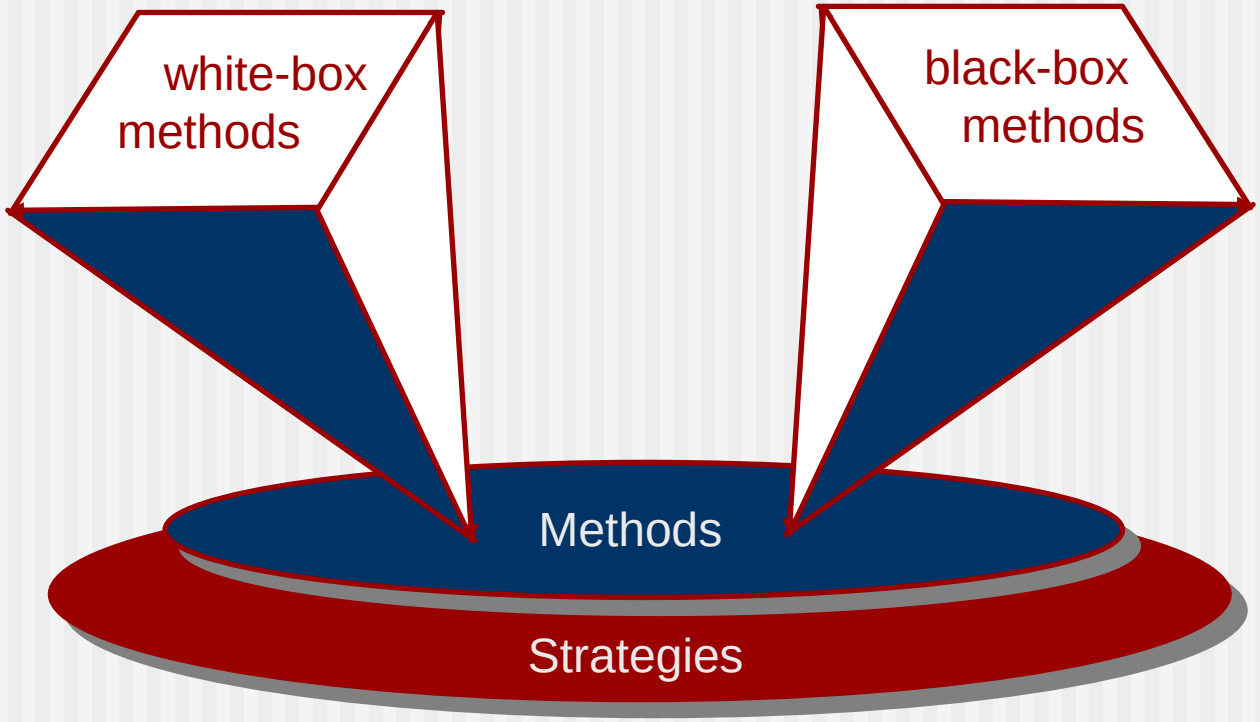


There are 10^{14} possible paths! If we execute one test per millisecond, it would take 3,170 years to test this program!!

Selective Testing



Software Testing



Model-Based Testing

- Analyze an existing behavioral model for the software or create one.
  - 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.

Model-Based Testing VS Model-Driven Test Design

- MBT Derive tests from a model of the software which are created during software development phases
  - such as a UML diagram
- But MDTD derive our tests from abstract structures.
  - These structures can be created *after* the software is implemented,

Model-Driven Test Design makes the distinctions between Black and White Box testing less important.

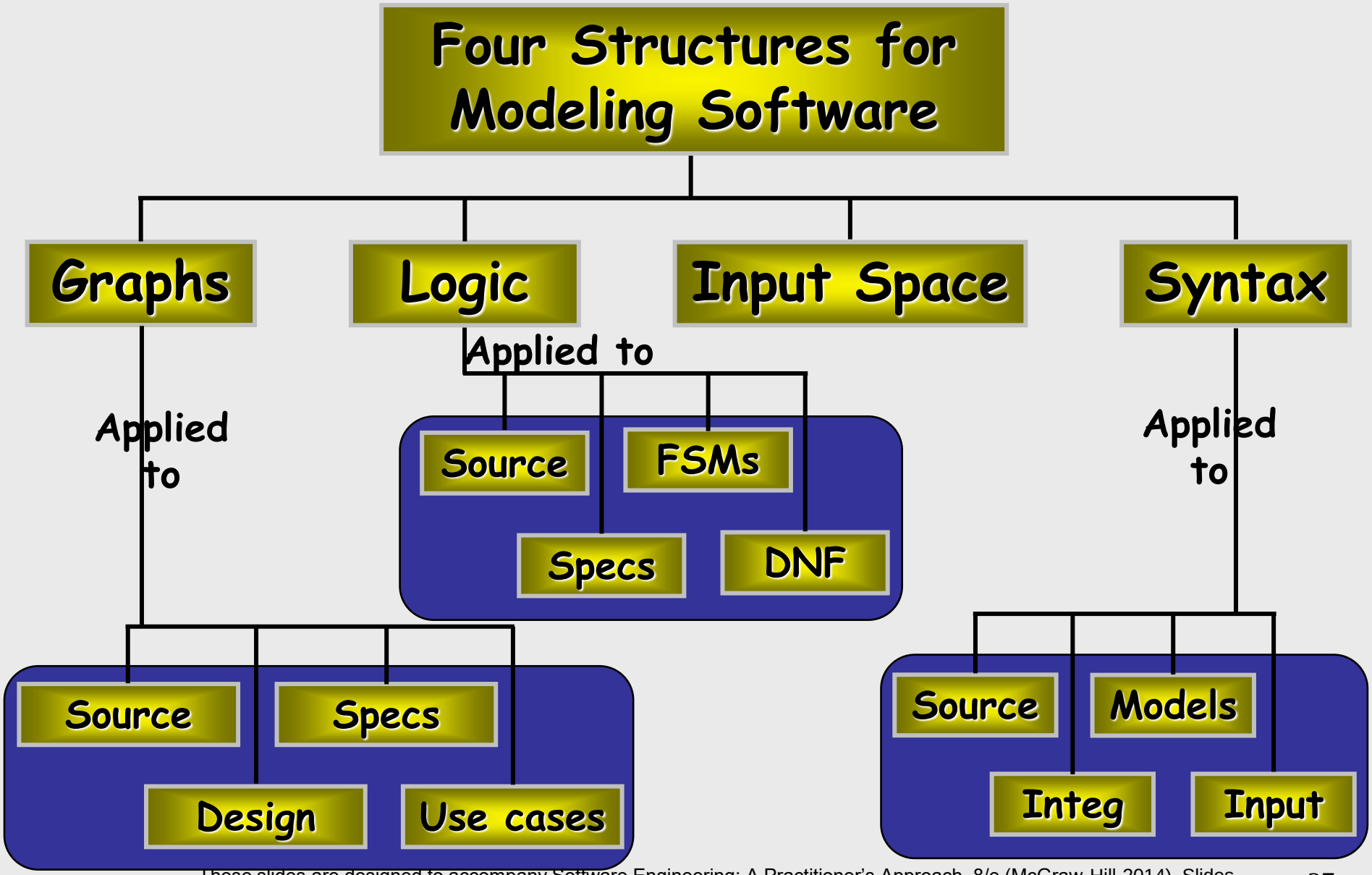
The more general question is:

from what abstraction level do we derive tests?

Model-Driven Test Design

- Model-Driven *Test Design*
  - is the process of designing input values that will effectively test software from abstract structures
  - is one of several activities for testing software
    - Most mathematical
    - Most technically challenging

Model Driven Testing: Coverage Overview



Deriving Test Cases from Graph

- Using the design or code as a foundation, draw a corresponding 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:Covering Graphs

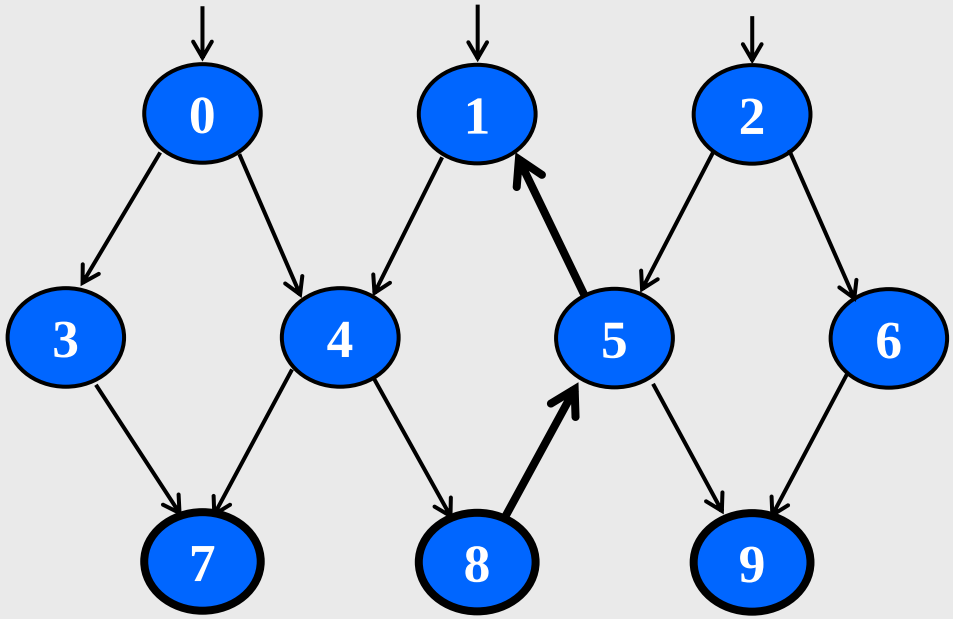
- Graphs are the most commonly used structure for testing
- Graphs can come from many sources
  - Control flow graphs
  - Design structure
  - FSMs and statecharts
  - Use cases
- Tests usually are intended to "cover" the graph in some way

Graph:Definition of a Graph

- A set *N* of nodes, *N* is not empty
- A set N_0 of initial nodes, N_0 is not empty
- A set N_f of final nodes, N_f is not empty
- A set *E* of edges, each edge from one node to another
  - (n_i , n_j), *i* is predecessor, *j* is successor

Graph: Paths in Graphs

- Path : A sequence of nodes – [n_1, n_2, …, n_M]
  - Each pair of nodes is an edge
- Length : The number of edges
  - A single node is a path of length 0
- Subpath : A subsequence of nodes in *p* is a subpath of *p*
- Reach (*n*) : Subgraph that can be reached from *n*



Paths

[0, 3, 7]

[1, 4, 8, 5, 1]

[2, 6, 9]

Reach (0) =

{ 0, 3, 4, 7, 8, 5, 1, 9 }

Reach ({0, 2}) = G

Reach([2,6]) =

{2, 6, 9}

Graph in Requirement: UML Use Cases

- UML use cases are often used to express software requirements
- They help express computer application workflow
- Use cases are commonly elaborated (or documented)
- Elaboration is first written textually
  - Details of operation
  - Alternatives model choices and conditions during execution

Graph in Requirement: Example: ATM Use Case

- Use Case Name : Withdraw Funds
- Summary : Customer uses a valid card to withdraw funds from a valid bank account.
- Actor : ATM Customer
- Precondition : ATM is displaying the idle welcome message
- Description :
  - Customer inserts an ATM Card into the ATM Card Reader.
  - If the system can recognize the card, it reads the card number.
  - System prompts the customer for a PIN.
  - Customer enters PIN.
  - System checks the card's expiration date and whether the card has been stolen or lost.
  - If the card is valid, the system checks if the entered PIN matches the card PIN.
  - If the PINs match, the system finds out what accounts the card can access.
  - System displays customer accounts and prompts the customer to choose a type of transaction. There are three types of transactions, Withdraw Funds, Get Balance and Transfer Funds. (The previous eight steps are part of all three use cases; the following steps are unique to the Withdraw Funds use case.)

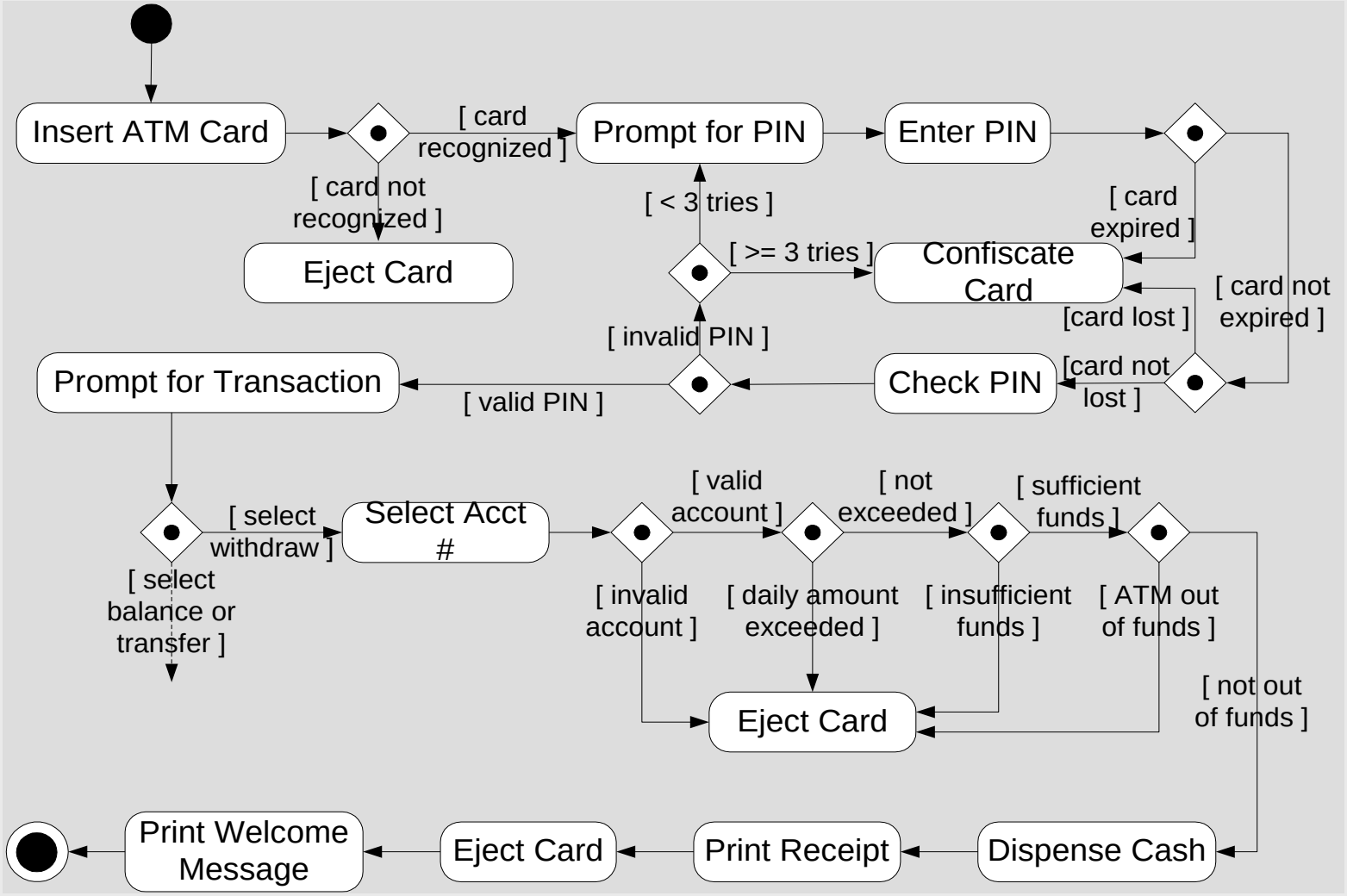
Graph in Requirement: Example: ATM Use Case – (2/3)

- Description (continued) :
  - Customer selects Withdraw Funds, selects the account number, and enters the amount.
  - System checks that the account is valid, makes sure that customer has enough funds in the account, makes sure that the daily limit has not been exceeded, and checks that the ATM has enough funds.
  - If all four checks are successful, the system dispenses the cash.
  - System prints a receipt with a transaction number, the transaction type, the amount withdrawn, and the new account balance.
  - System ejects card.
  - System displays the idle welcome message.

Graph in Requirement: Example: ATM Use Case – (3/3)

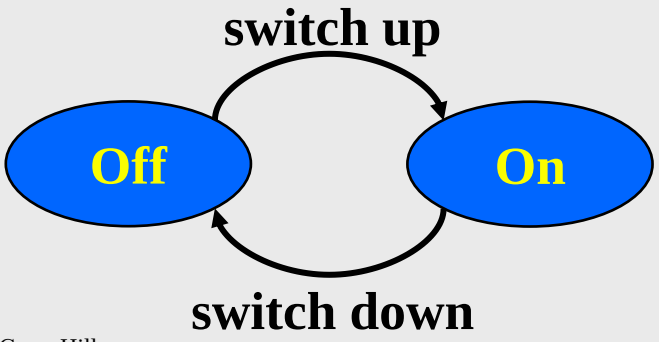
- Alternatives :
  - If the system cannot recognize the card, it is ejected and the welcome message is displayed.
  - If the current date is past the card's expiration date, the card is confiscated and the welcome message is displayed.
  - If the card has been reported lost or stolen, it is confiscated and the welcome message is displayed.
  - If the customer entered PIN does not match the PIN for the card, the system prompts for a new PIN.
  - If the customer enters an incorrect PIN three times, the card is confiscated and the welcome message is displayed.
  - If the account number entered by the user is invalid, the system displays an error message, ejects the card and the welcome message is displayed.
  - If the request for withdraw exceeds the maximum allowable daily withdrawal amount, the system displays an apology message, ejects the card and the welcome message is displayed.
  - If the request for withdraw exceeds the amount of funds in the ATM, the system displays an apology message, ejects the card and the welcome message is displayed.
  - If the customer enters Cancel, the system cancels the transaction, ejects the card and the welcome message is displayed.
- Postcondition :
  - Funds have been withdrawn from the customer's account.

Graph in Requirement: ATM Withdraw Activity Graph



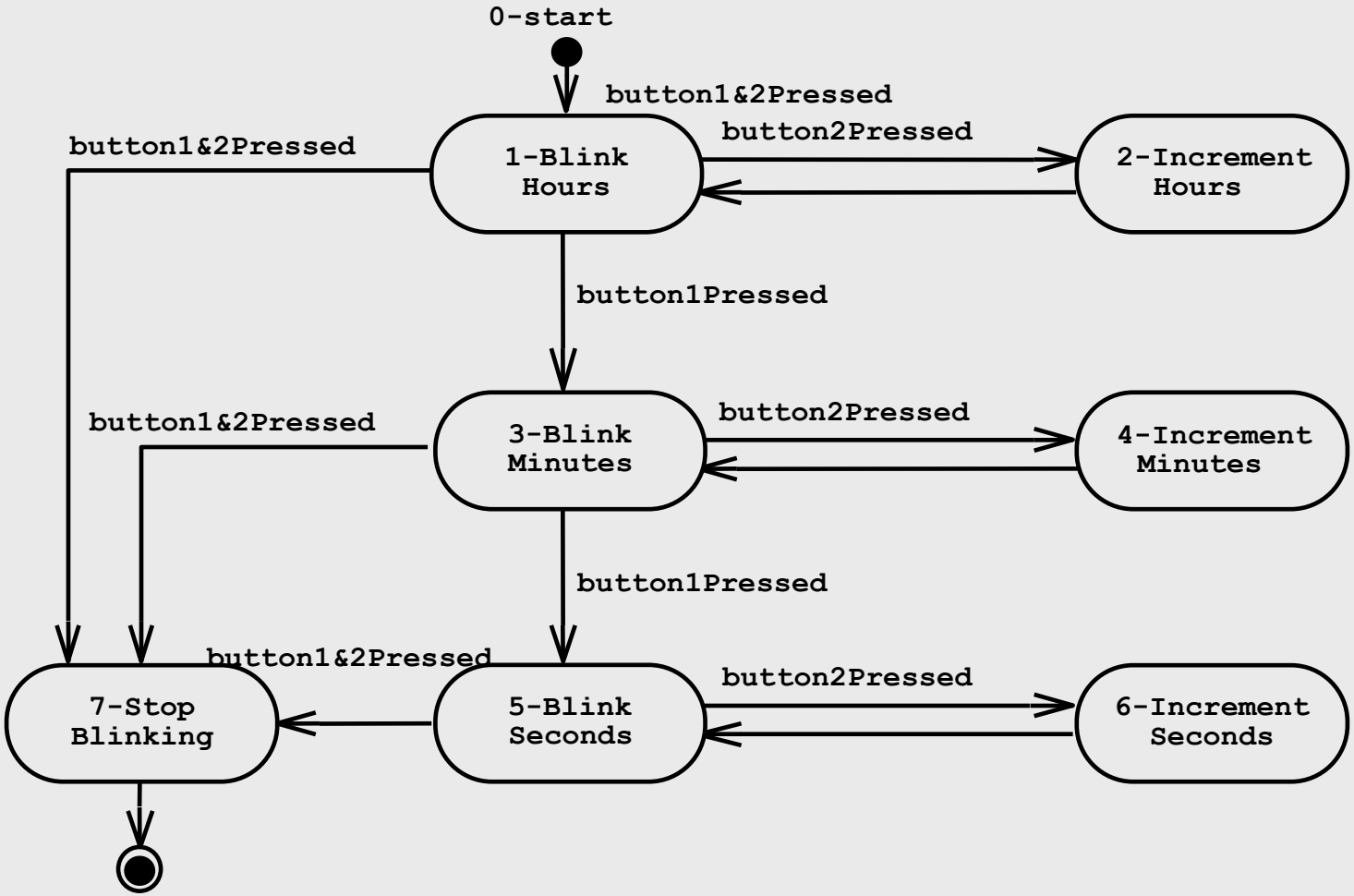
Graph in Design

- A design model describes aspects of what behavior software should exhibit
- State behavior descriptions of software
- A finite state machine (FSM) is a graph that describes how software variables are modified during execution
- Nodes : States, representing sets of values for key variables
- Edges : Transitions, possible changes in the state



Graph in Design:

Simple Watch: Using Implicit or Explicit Specifications



A UML statechart diagram for `SetTime` use case of the `SimpleWatch`.

Graph in Source: Control Flow Graphs

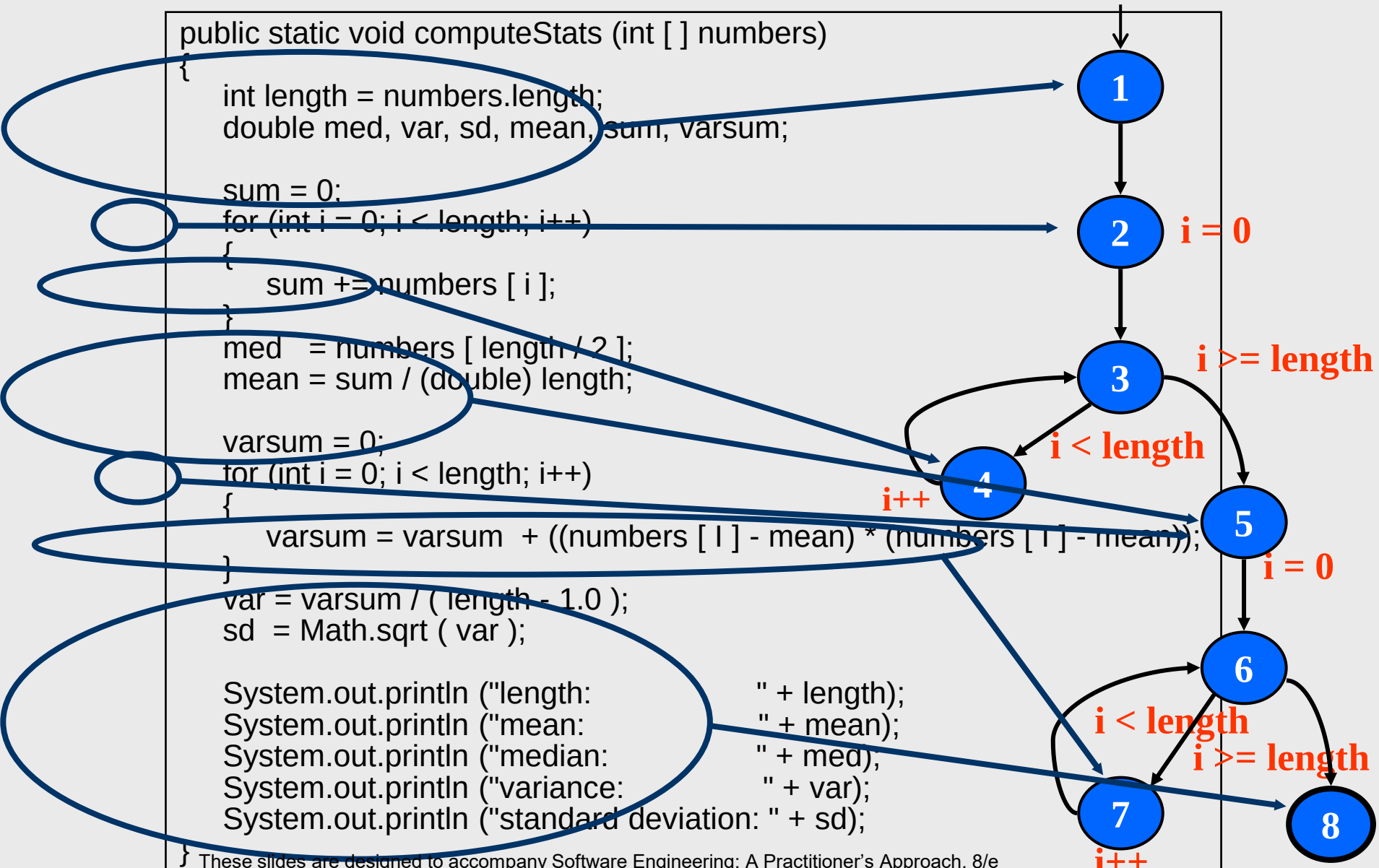
- The most common application of graph criteria is to program source
- Graph : Usually the control flow graph (CFG)
- A CFG models all executions of a method by describing control structures
- Nodes : Statements or sequences of statements (basic blocks)
- Edges : Transfers of control
- Basic Block : A sequence of statements such that if the first statement is executed, all statements will be (no branches)
- CFGs are sometimes annotated with extra information
  - branch predicates
  - defs
  - uses

Graph in Source: Example Control Flow – Stats

```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2 ];
    mean = sum / (double) length;

    varsum = 0;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ I ] - mean) * (numbers [ I ] -
mean));   }
    var = varsum / ( length - 1.0 );
    sd  = Math.sqrt ( var );
    System.out.println ("length:                " + length);
    System.out.println ("mean:                  " + mean);
    System.out.println ("median:                " + med);
    System.out.println ("variance:               " + var);
    System.out.println ("standard deviation: " + sd);
}
```

Graph in Source: Example Control Flow – Stats

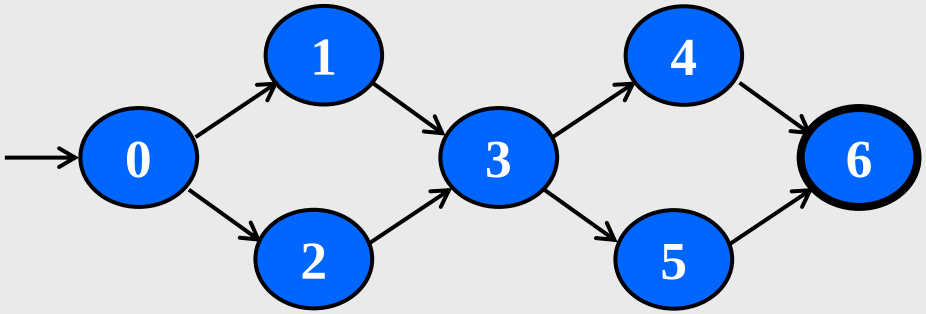
```
public static void computeStats (int [ ] numbers)
{
    int length = numbers.length;
    double med, var, sd, mean, sum, varsum;

    sum = 0;
    for (int i = 0; i < length; i++)
    {
        sum += numbers [ i ];
    }
    med   = numbers [ length / 2 ];
    mean = sum / (double) length;

    varsum = 0;
    for (int i = 0; i < length; i++)
    {
        varsum = varsum  + ((numbers [ I ] - mean) * (numbers [ I ] - mean));
    }
    var = varsum / ( length - 1.0 );
    sd  = Math.sqrt ( var );

    System.out.println ("length:                " + length);
    System.out.println ("mean:                  " + mean);
    System.out.println ("median:                " + med);
    System.out.println ("variance:              " + var);
    System.out.println ("standard deviation: " + sd);
}
```



Graph: Test Paths and SESEs

- Test Path : A path that starts at an initial node and ends at a final node
- Test paths represent execution of test cases
  - Some test paths can be executed by many tests
  - Some test paths cannot be executed by any tests

- SESE graphs : All test paths start at a single node and end at another node
  - Single-entry, single-exit
  - N0 and Nf have exactly one node



Double-diamond graph

Four test paths

[0, 1, 3, 4, 6]

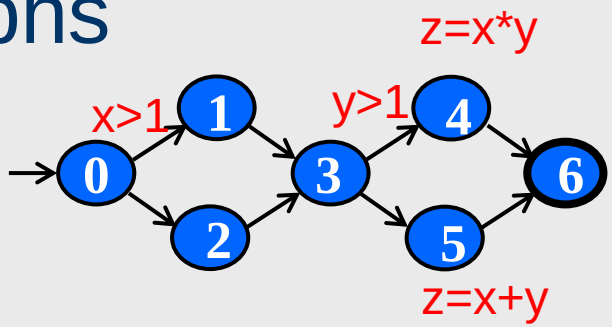
[0, 1, 3, 5, 6]

[0, 2, 3, 4, 6]

[0, 2, 3, 5, 6]

Graph: Testing and Covering Graphs

- Test Requirements (TR) :
  - Describe properties of test paths
- Test Criterion :
  - Rules that define test requirements
- Satisfaction : Given
  - A graph G,
  - a graph test criterion C,
  - a set TR of test requirements for C,
  - a set of test path TP
  - a test set T satisfies C on graph G
    - if and only if for every test requirement tr in TR,
    - there is a test path p in TP such that p meets tr.



C: Node Coverage
TR={0,1,2,3,4,5,6}
TP={[0,1,3,4,6],
[0,2,3,5,6]}
T={(x=5,y=3,z=?),
(x=-5,y=-3,z=?)}
tr=4
p=[0,1,3,4,6]

Graph: Node and Edge Coverage

- The first (and simplest) two criteria require that each node and edge in a graph be executed

Node Coverage (NC) : Test set *T* satisfies node coverage on graph *G* iff for every syntactically reachable node *n* in *N*, there is some path *p* in *TP* such that *p* visits *n*.

- **This statement is a bit cumbersome, so we abbreviate it in terms of the set of test requirements**

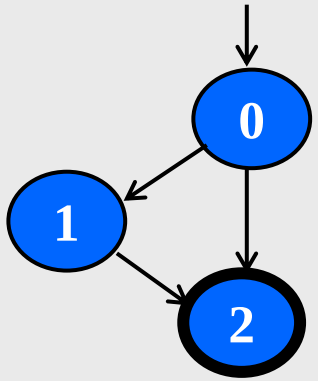
Node Coverage (NC) : TR contains each reachable node in G.

Graph: Node and Edge Coverage

- Edge coverage is slightly stronger than node coverage

Edge Coverage (EC) : TR contains each reachable edge (path of length 1) in G.

- **NC and EC are only different when there is an edge and another subpath between a pair of nodes (as in an "if-else" statement)**

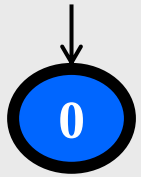


Node Coverage : TR = { 0, 1, 2 }
Test Path = [0, 1, 2]

Edge Coverage : TR = { (0,1), (0, 2), (1, 2) }
Test Paths = [0, 1, 2]
[0, 2]

Graph: Paths of Length 1 and 0

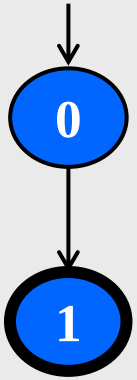
- A graph with only one node will not have any edges



- It may be boring, but formally, Edge Coverage needs to require Node Coverage on this graph
- Otherwise, Edge Coverage will not subsume Node Coverage
  So we define "length up to 1" instead of simply "length 1"

Edge Coverage (EC) : TR contains each reachable path of length up to 1, inclusive, in G.

- We have the same issue with graphs that only have one edge – for Edge Pair Coverage …



Graph: Covering Multiple Edges

- Edge-pair coverage requires pairs of edges, or subpaths of length 2

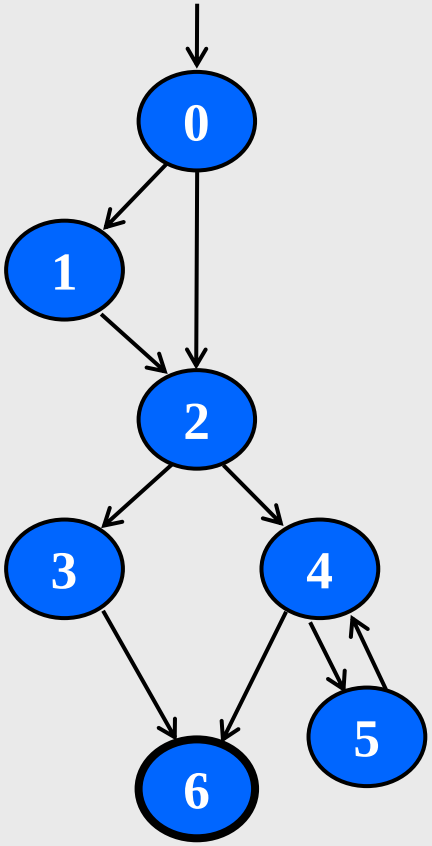
Edge-Pair Coverage (EPC) : TR contains each reachable path of length up to 2, inclusive, in G.

- **The "length up to 2" is used to include graphs that have less than 2 edges**
- **The logical extension is to require all paths …**

Complete Path Coverage (CPC) : TR contains all paths in G.

- **Unfortunately, this is impossible if the graph has a loop,**

Graph: Structural Coverage Example



Node Coverage

TR = { 0, 1, 2, 3, 4, 5, 6 }

Test Paths: [0, 1, 2, 3, 6] [0, 1, 2, 4, 5, 4, 6]

Edge Coverage

TR = { (0,1), (0,2), (1,2), (2,3), (2,4), (3,6), (4,5), (4,6), (5,4) }

Test Paths: [0, 1, 2, 3, 6] [0, 2, 4, 5, 4, 6]

Edge-Pair Coverage

TR = { [0,1,2], [0,2,3], [0,2,4], [1,2,3], [1,2,4], [2,3,6], [2,4,5], [2,4,6], [4,5,4], [5,4,5], [5,4,6] }

Test Paths: [0, 1, 2, 3, 6] [0, 1, 2, 4, 6] [0, 2, 3, 6] [0, 2, 4, 5, 4, 5, 4, 6]

Complete Path Coverage

Test Paths: [0, 1, 2, 3, 6] [0, 1, 2, 4, 6] [0, 1, 2, 4, 5, 4, 6] [0, 1, 2, 4, 5, 4, 5, 4, 6] [0, 1, 2, 4, 5, 4, 5, 4, 5, 4, 6] …

Graph: Simple Paths and Prime Paths

- Simple Path : A path from node ni to nj is simple if no node appears more than once, except possibly the first and last nodes are the same
  - No internal loops
  - Includes all other subpaths
  - A loop is a simple path
- Prime Path : A simple path that does not appear as a proper subpath of any other simple path

Control Structure Testing

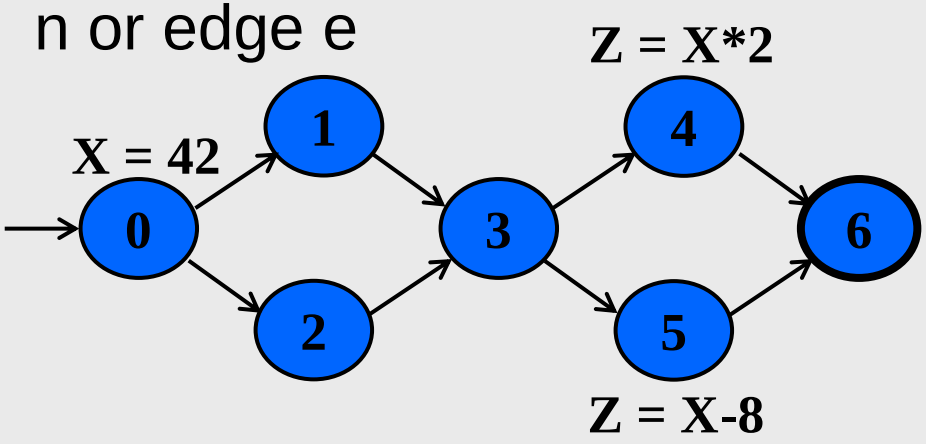
- Data flow testing
  - selects test paths of a program according to the locations of definitions and uses of variables in the program

Data Flow Testing

- The data flow testing method selects test paths of a program according to the locations of definitions and uses of variables.

Graph: Data Flow Criteria

- Definition (def) : A location where a value for a variable is stored into memory
- Use : A location where a variable's value is accessed
- def (n) or def (e) : The set of variables that are defined by node n or edge e
- use (n) or use (e) : The set of variables that are used by node n or edge e



Defs: def (0) = {X}

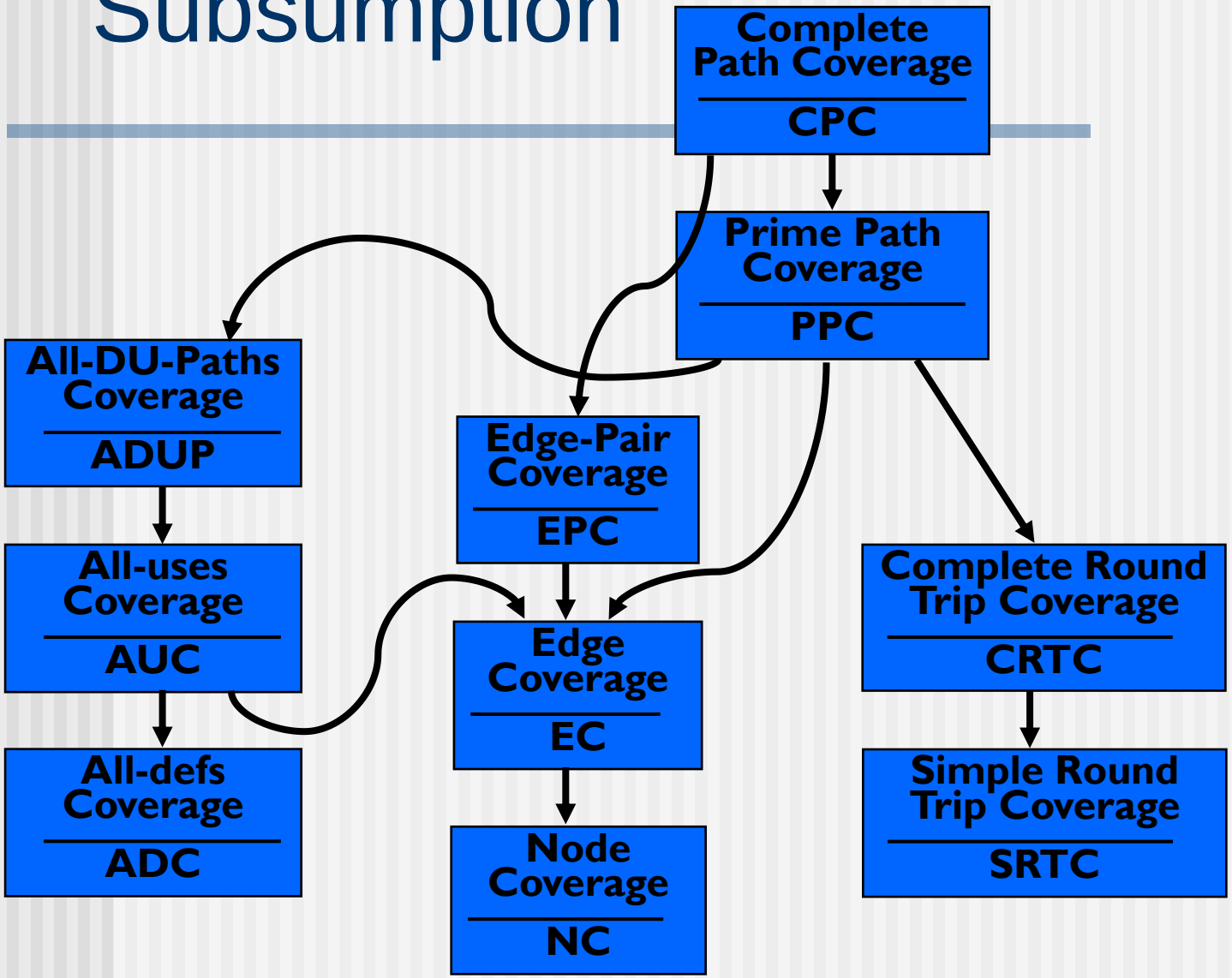
def (4) = {Z}

def (5) = {Z}

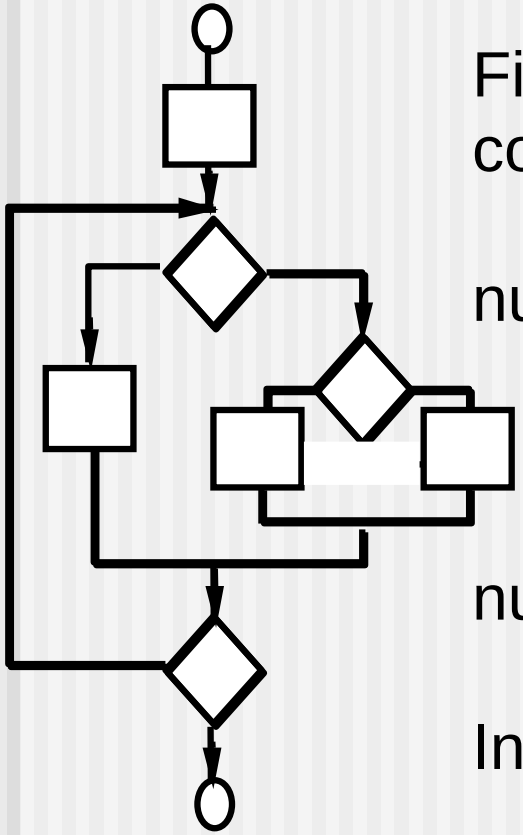
Uses: use (4) = {X}

use (5) = {X}

Graph Coverage Criteria Subsumption



Basis Path Testing



First, we compute the cyclomatic complexity:

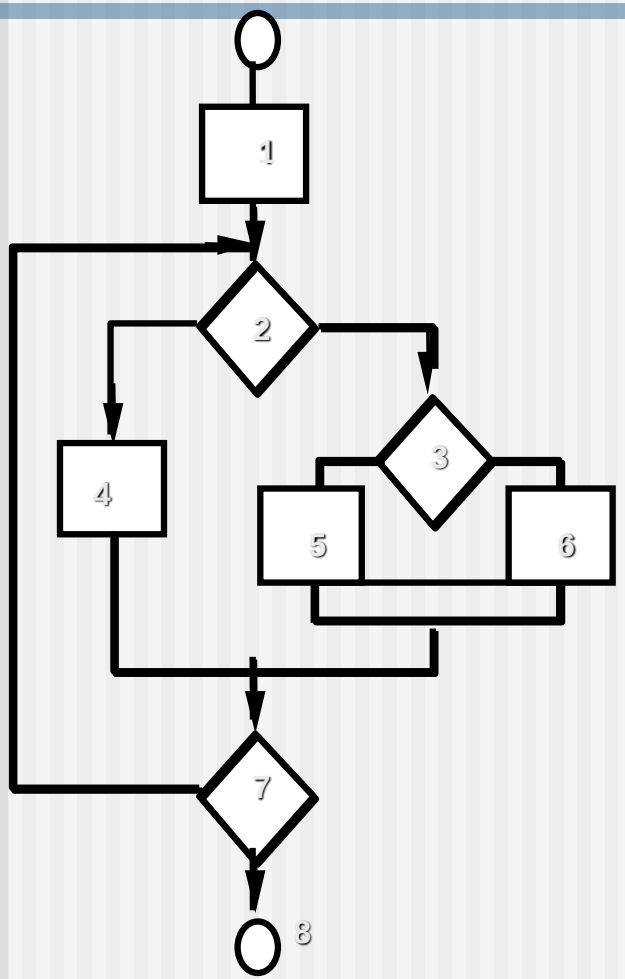
number of simple decisions + 1

or

number of enclosed areas + 1

In this case, V(G) = 4

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

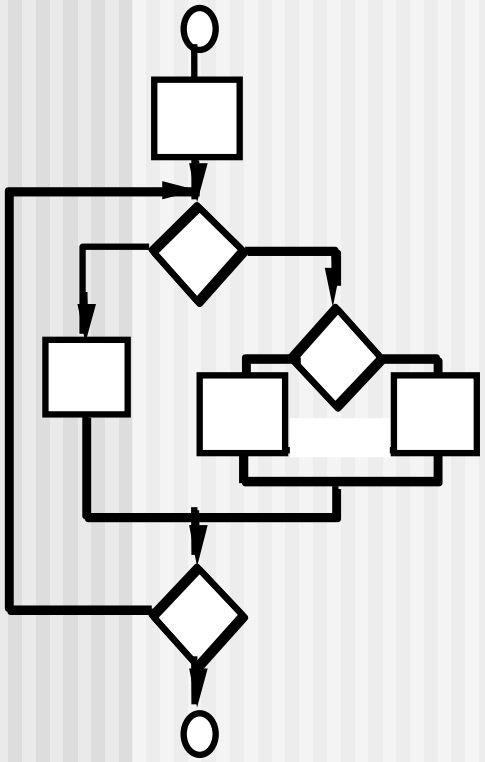
Path 2: 1,2,3,5,7,8

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

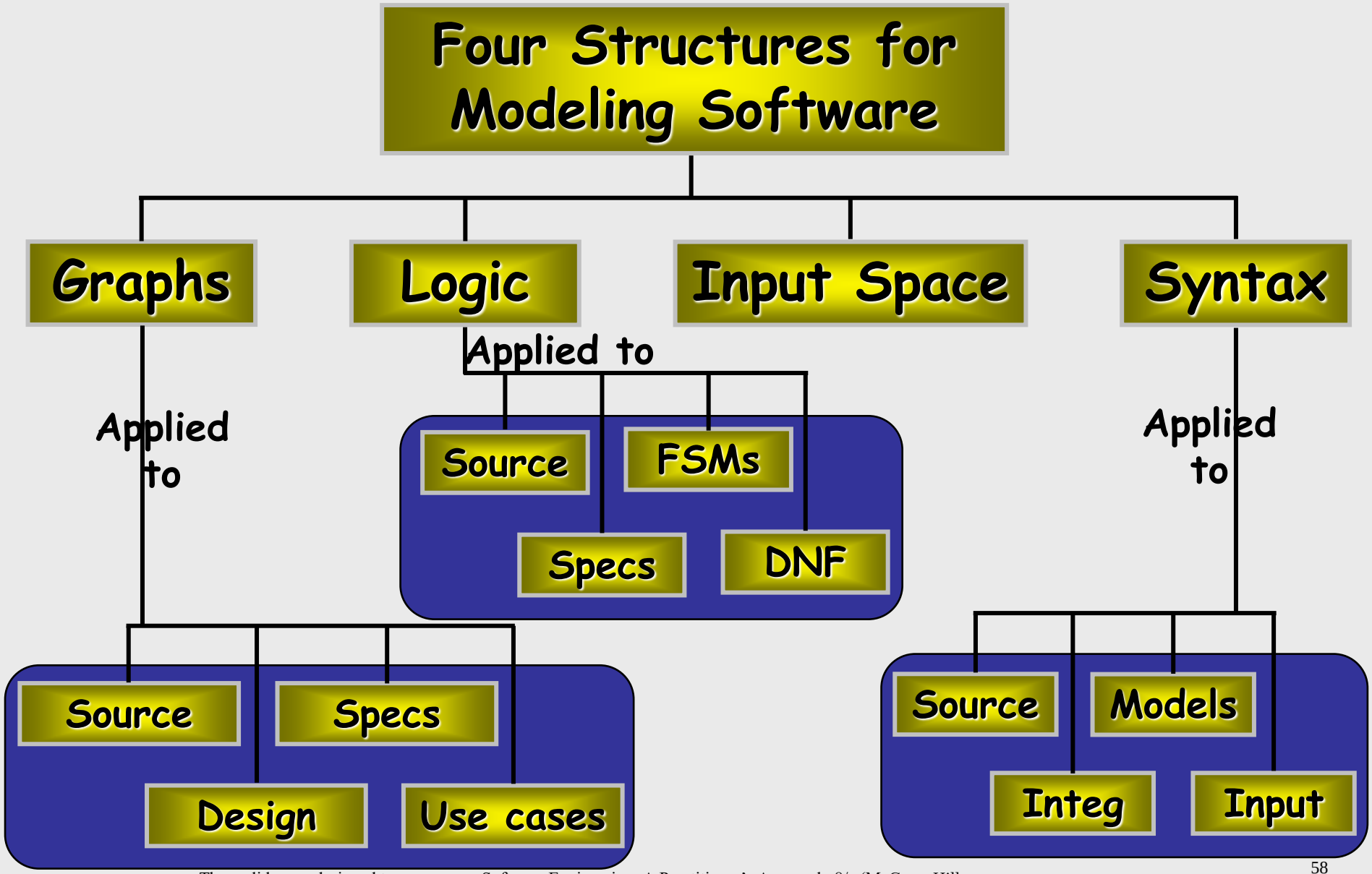


- you don't need a flow chart, but the picture will help when you trace program paths
- count each simple logical test, compound tests count as 2 or more
- basis path testing should be applied to critical modules

Control Structure Testing

- Condition testing
  - a test case design method that exercises the logical conditions contained in a program module

Model Driven Testing: Coverage Overview



Logic: Logic Expressions

- Logical expressions can come from many sources
  - Decisions in programs
  - FSMs and statecharts
  - Requirements
- Tests are intended to choose some subset of the total number of truth assignments to the expressions
- Covering logic expressions is required by the US Federal Aviation Administration for safety critical software

Logic:
Logic Predicates and Clauses

- A *predicate* is an expression that evaluates to a boolean value
- Predicates can contain
  - boolean variables
  - non-boolean variables that contain >, <, ==, >=, <=, !=
  - boolean function calls
- Internal structure is created by logical operators
  - $\neg$ – the *negation* operator
  - $\wedge$ – the *and* operator
  - $\vee$ – the *or* operator
  - $\rightarrow$ – the *implication* operator
  - $\oplus$ – the *exclusive or* operator
  - $\leftrightarrow$ – the *equivalence* operator
- A *clause* is a predicate with no logical operators

Logic: Examples

- (a < b) ∨ f (z) ∧ D ∧ (m >= n*o)
- Four clauses:
  - (a < b) – relational expression
  - f (z) – boolean-valued function
  - D – boolean variable
  - (m >= n*o) – relational expression
- Most predicates have few clauses
  - It would be nice to quantify that claim!
- Sources of predicates
  - Decisions in programs
  - Guards in finite state machines
  - Decisions in UML activity graphs
  - Requirements, both formal and informal
  - SQL queries

Logic in Requirements: Translating from English

- "I am interested in SWE 637 and CS 652"
- *course = swe637 OR course = cs652*

Humans have trouble translating from English to Logic

- "If you leave before 6:30 AM, take Braddock to 495, if you leave after 7:00 AM, take Prosperity to 50, then 50 to 495"
- *time < 6:30 → path = Braddock ∧ time > 7:00 → path = Prosperity*
- Hmm … this is incomplete !
- *time < 6:30 → path = Braddock ∧ time >= 6:30 → path = Prosperity*

Logic in Requirements: Specifications in Software

- Specifications can be formal or informal
  - Formal specs are usually expressed mathematically
  - Informal specs are usually expressed in *natural language*
- Lots of formal languages and informal styles are available

- Most specification languages include explicit logical expressions, so it is very easy to apply logic coverage criteria

- Implicit logical expressions in natural-language specifications should be re-written as explicit logical expressions as part of test design
  - You will often find mistakes
- One of the most common is preconditions …

Logic in Requirements: Preconditions

- Programmers often include preconditions for their methods
- The preconditions are often expressed in comments in method headers
- Preconditions can be in javadoc, “requires”, “pre”, …

Example – Saving addresses

// name must not be empty
// state must be valid
// zip must be 5 numeric digits
// street must not be empty
// city must not be empty

Rewriting to logical expression

name != “” ∧ state in stateList ∧ zip >= 00000 ∧ zip <= 99999 ∧ street != “” ∧ city != “”

Logic in Requirements: Example 1: Valve System

- Consider a system with a valve that might be either open or closed, and several modes, two of which are "Operational" and "Standby."
- This leads to the following clause definitions:

  a = "*The valve is closed*"

  b = "*The system status is Operational*"

  c = "*The system status is Standby*"
- Suppose that a certain action can be taken only if the valve is closed and the system status is either in Operational or Standby. That is:

  p = *valve is closed AND (system status is Operational OR system status is Standby)*

  $= a \wedge (b \vee c)$

Logic in Requirements: Example 2-DIV2

- // if x <= 1 then q= 0
- // else compute q(quotient), of division of x by 2
- 1. Procedure DIV2 (x:int) return(q:int)
- 2. { q:=0;
- 3. do while (x > 1)
- 4. q:= q+1;
- 5. x:= x - 2;
- 6. end
- 7. return(q)
- 8. }

Formal Specification :

$$\forall\, x : \mathbb{Z} \bullet \exists\, q : \mathbb{N} \bullet (\underbrace{x \leq 1}_{a} \wedge \underbrace{q = 0}_{b}) \vee (\underbrace{x > 1}_{\neg\, a} \wedge (\underbrace{x = 2 * q}_{c} \vee \underbrace{x = 2 * q + 1}_{d}))$$

$$p : (a \wedge b) \vee (\neg\, a \wedge (c \vee d))$$

Logic in Requirements: Example 3-Calculate no of Days

```
public static int cal (int month1, int day1, int month2,
                       int day2, int year)
{
//*************************************************************
// Calculate the number of Days between the two given days in
// the same year.
// preconditions : day1 and day2 must be in same year
//              1 <= month1, month2 <= 12
//              1 <= day1, day2 <= 31
//              month1 <= month2
//              The range for year: 1 ... 10000
//*************************************************************
   int numDays;

   if (month2 == month1) // in the same month
      numDays  = day2 - day1;
   else
   {

// Skip month 0.
      int daysIn[] = {0, 31, 0, 31, 30, 31, 30, 31, 31,
30, 31, 30, 31};
      // Are we in a leap year?
      int m4 = year % 4;
      int m100 = year % 100;
      int m400 = year % 400;
      if ((m4 != 0) || ((m100 == 0) && (m400 != 0)))
         daysIn[2] = 28;
      else
         daysIn[2] = 29;

      // start with days in the two months
      numDays = day2 + (daysIn[month1] - day1);

      // add the days in the intervening months
      for (int i = month1 + 1; i <= month2-1; i++)
         numDays = daysIn[i] + numDays;
   }
   return (numDays);
}
```

Logic in Requirements: Example 3-Calculate no of Days

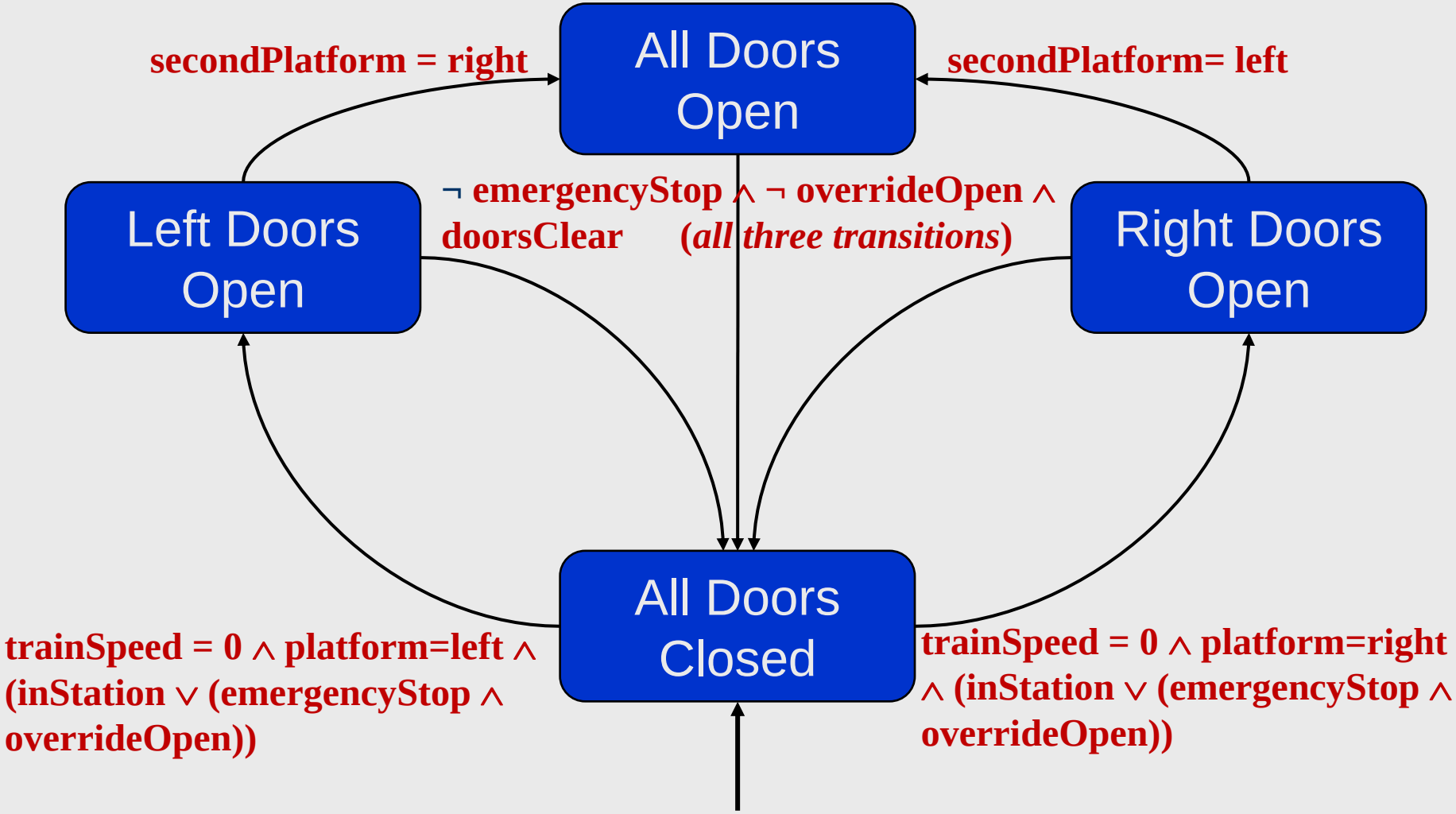
- Month1 >= 1 /\
- Month1 <=12 /\
- Month2 >= 1 /\
- Month2 <=12 /\
- Month1 <= Month2 /\
- Day1 >= 1 /\
- Day1 <= 31 /\
- Day2 >= 1 /\
- Day2 <= 31 /\
- Year >= 1 /\
- Year <= 10000

Logic in Design: Finite State Machines

- FSMs are graphs
  - nodes represent state
  - edges represent transitions among states
- Transitions often have logical expressions as guards or triggers
- As we said:

Find a *logical expression* and cover it

Logic in Design: Finite State Machines
Example 4– Subway Train



Logic in Source

- Predicates are derived from decision statements in programs
- In programs, most predicates have less than four clauses
  - Wise programmers actively strive to keep predicates simple
- Applying logic criteria to program source is hard because of reachability and controllability:
  - *Reachability* : Before applying the criteria on a predicate at a particular statement, we have to get to that statement
  - *Controllability* : We have to find input values that indirectly assign values to the variables in the predicates
  - Variables in the predicates that are not inputs to the program are called *internal variables*

Logic in Source : Example 5-Triang *(pg 1 of 5)*

```
 1  // Jeff Offutt -- Java version Feb 2003
 2  // The old standby: classify triangles
 3  // Figures 3.2 and 3.3 in the book.
 4  import java.io.*;
 5  class trityp
 6  {
 7    private static String[] triTypes = { "",    // Ignore 0.
 8          "scalene", "isosceles", "equilateral", "not a valid
          triangle"};
 9   private static String instructions = "This is the ancient
    TriTyp program.\nEnter three integers that represent the lengths
    of the sides of a triangle.\nThe triangle will be categorized as
    either scalene, isosceles, equilateral\nor invalid.\n";
10
11  public static void main (String[] argv)
12  {  // Driver program for trityp
13    int A, B, C;
14    int T;
```

Logic in Source
Example 5-Triang *(pg 2 of 5)*

```
16    System.out.println (instructions);
17    System.out.println ("Enter side 1: ");
18    A = getN();
19    System.out.println ("Enter side 2: ");
20    B = getN();
21    System.out.println ("Enter side 3: ");
22    C = getN();
23    T = Triang (A, B, C);
24
25    System.out.println ("Result is: " + triTypes [T]);
26  }
27
28  // ====================================
```

Logic in Source
Example 5-Triang *(pg 3 of 5)*

```
29  // The main triangle classification method
30  private static int Triang (int Side1, int Side2, int Side3)
31  {
32     int tri_out;
33
34     // tri_out is output from the routine:
35     //    Triang = 1 if triangle is scalene
36     //    Triang = 2 if triangle is isosceles
37     //    Triang = 3 if triangle is equilateral
38     //    Triang = 4 if not a triangle
39
40     // After a quick confirmation that it's a legal
41     // triangle, detect any sides of equal length
42     if (Side1 <= 0 || Side2 <= 0 || Side3 <= 0)
43     {
44        tri_out = 4;
45        return (tri_out);
46     }
```

Logic in Source : Example 5-Triang *(pg 4 of 5)*

```
48     tri_out = 0;
49    if (Side1 == Side2)
50       tri_out = tri_out + 1;
51    if (Side1 == Side3)
52       tri_out = tri_out + 2;
53    if (Side2 == Side3)
54       tri_out = tri_out + 3;
55    if (tri_out == 0)
56    {  // Confirm it's a legal triangle before declaring
57       // it to be scalene
58
59       if (Side1+Side2 <= Side3 || Side2+Side3 <= Side1 ||
60           Side1+Side3 <= Side2)
61          tri_out = 4;
62       else
63          tri_out = 1;
64       return (tri_out);
65    }
```

Logic in Source
Example 5-Triang *(pg 5 of 5)*

```
67    /* Confirm it's a legal triangle before declaring  */
68    /* it to be isosceles or equilateral  */
69
70    if (tri_out > 3)
71       tri_out = 3;
72    else if (tri_out == 1 && Side1+Side2 > Side3)
73      tri_out = 2;
74    else if (tri_out == 2 && Side1+Side3 > Side2)
75      tri_out = 2;
76    else if (tri_out == 3 && Side2+Side3 > Side1)
77      tri_out = 2;
78    else
79      tri_out = 4;
80    return (tri_out);
81  } // end Triang
```

Logic in Source : Example 5-Ten Triang Predicates

42: (Side1 <= 0 || Side2 <= 0 || Side3 <= 0)

49: (Side1 == Side2)

51: (Side1 == Side3)

53: (Side2 == Side3)

55: (triOut == 0)

59: (Side1+Side2 <= Side3 || Side2+Side3 <= Side1 ||
Side1+Side3 <= Side2)

70: (triOut > 3)

72: (triOut == 1 && Side1+Side2 > Side3)

74: (triOut == 2 && Side1+Side3 > Side2)

76: (triOut == 3 && Side2+Side3 > Side1)

Logic:
Testing and Covering Predicates(3.2)

- We use predicates in testing as follows :
  - Developing a model of the software as one or more predicates
  - Requiring tests to satisfy some combination of clauses

- Abbreviations:
  - *P* is the set of predicates
  - *p* is a single predicate in *P*
  - *C* is the set of clauses in *P*
  - C_p is the set of clauses in predicate *p*
  - *c* is a single clause in *C*

Logic: Predicate and Clause Coverage

- The first (and simplest) two criteria require that each predicate and each clause be evaluated to both true and false

Predicate Coverage (PC) : For each *p* in *P*, *TR* contains two requirements: *p* evaluates to true, and *p* evaluates to false.

- When predicates come from conditions on edges, this is equivalent to edge coverage
- PC does not evaluate all the clauses, so …

Clause Coverage (CC) : For each *c* in *C*, *TR* contains two requirements: *c* evaluates to true, and *c* evaluates to false.

Logic: Predicate and Clause Coverage

Predicate Coverage Example

((a < b) ∨ D) ∧ (m >= n*o)

Predicate = true

a = 5, b = 10, D = true, m = 1, n = 1, o = 1

= (5 < 10) ∨ true ∧ (1 >= 1*1)

= true ∨ true ∧ TRUE

= true

Predicate = false

a = 10, b = 5, D = false, m = 1, n = 1, o = 1

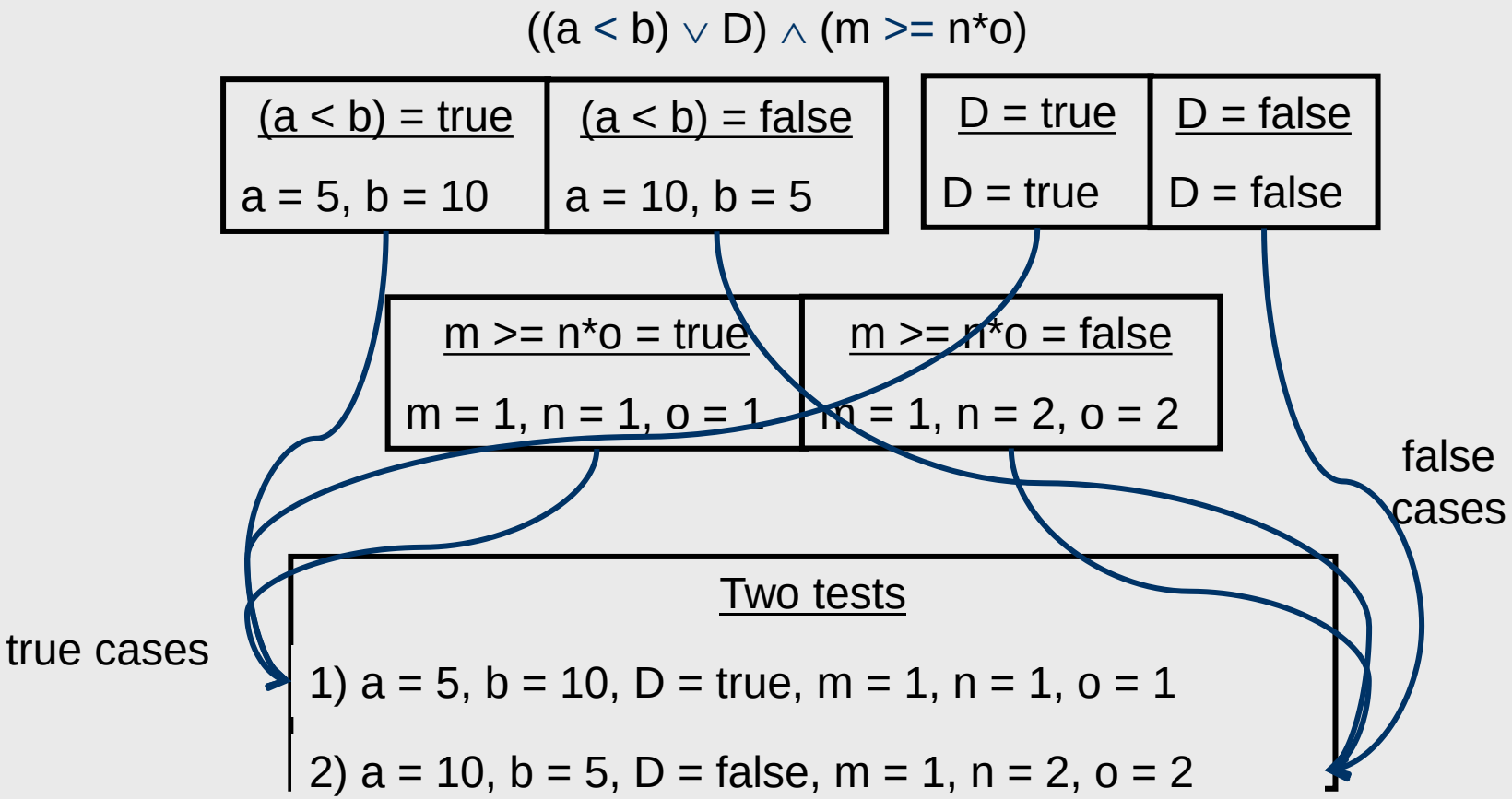
= (10 < 5) ∨ false ∧ (1 >= 1*1)

= false ∨ false ∧ TRUE

= false

Logic: Predicate and Clause Coverage

Clause Coverage Example



Logic: Predicate and Clause Coverage

Problems with PC and CC

- PC does not fully exercise all the clauses, especially in the presence of short circuit evaluation
- CC does not always ensure PC
  - That is, we can satisfy CC without causing the predicate to be both true and false
  - This is definitely not what we want !
- The simplest solution is to test all combinations …

Logic: Combinatorial Coverage

- CoC requires every possible combination
- Sometimes called Multiple Condition Coverage

Combinatorial Coverage (CoC) : For each p in P, TR has test requirements for the clauses in Cp to evaluate to each possible combination of truth values.

	a < b	D	m >= n*o	((a < b) ∨ D) ∧ (m >= n*o)
1	T	T	T	T
2	T	T	F	F
3	T	F	T	T
4	T	F	F	F
5	F	T	T	T
6	F	T	F	F
7	F	F	T	F
8	F	F	F	F

Logic: Combinatorial Coverage

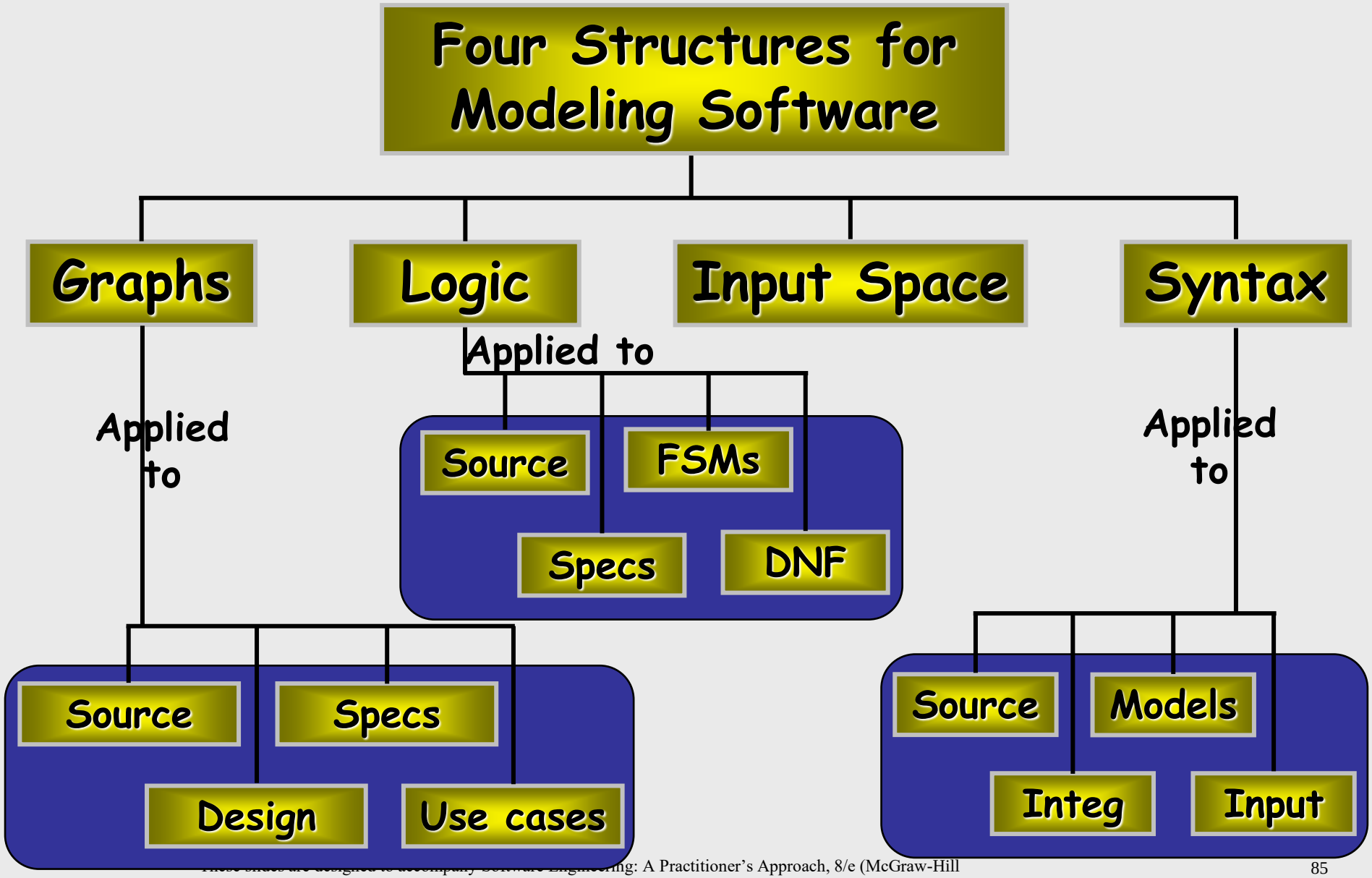
- This is simple, neat, clean, and comprehensive …

- But quite expensive!
- 2^N tests, where N is the number of clauses
  - Impractical for predicates with more than 3 or 4 clauses
- The literature has lots of suggestions – some confusing
- The general idea is simple:

Test each clause independently from the other clauses

- Getting the details right is hard
- What exactly does "independently" mean ?
- The book presents this idea as "*making clauses active*" …

Model Driven Testing: Coverage Overview



Input Domains

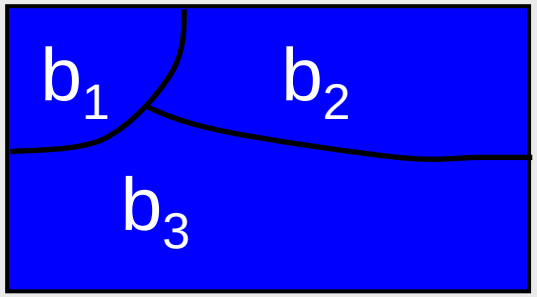
- The input domain to a program contains all the possible inputs to that program
- For even small programs, the input domain is so large that it might as well be infinite
- Testing is fundamentally about choosing finite sets of values from the input domain
- *Input parameters* define the scope of the input domain
  - Parameters to a method
  - Data read from a file
  - Global variables
  - User level inputs
- Domain for each input parameter is partitioned into regions
- At least one value is chosen from each region

Benefits of ISP

- Can be equally applied at several levels of testing
  - Unit
  - Integration
  - System
- Relatively easy to apply with no automation
- Easy to adjust the procedure to get more or fewer tests
- No implementation knowledge is needed
  - just the input space

Partitioning Domains

- Partition
- Domain *D*
- Partition scheme *q* of *D*
- The partition *q* defines a set of blocks, $Bq = b_1, b_2, \ldots b_Q$
- The partition must satisfy two properties:
  1. blocks must be pairwise disjoint (no overlap)
  2. together the blocks cover the domain *D* (complete)



$$b_i \cap b_j = \Phi, \forall\ i \neq j,\ b_i, b_j \in B_q$$

$$\bigcup_{b \in Bq} b = D$$

Using Partitions – Assumptions

- Choose a value from each block
- Each value is assumed to be equally useful for testing
- Application to testing
  - Find characteristics in the inputs : parameters, semantic descriptions, …
  - Partition each characteristics(into blocks)
  - Choose tests by combining values from characteristics (blocks)
- Characteristics
  - Input X is null
  - Order of the input file F (sorted, inverse sorted, arbitrary, …)
  - Min separation of two aircraft
  - Input device (DVD, CD, VCR, computer, …)

Choosing Partitions

- Choosing (or defining) partitions seems easy, but is easy to get wrong
- Consider the "*q: order of file F*"

b_1 = sorted in ascending order

b_2 = sorted in descending order

b_3 = arbitrary order

but … something's fishy …

What if the file is of length 1 or 0?

Recursive Insertion sort.

Base Case:

If array size is 1 or smaller, return.

Inductive step:

Recursively sort first n-1 elements.

Insert last element at its correct position in sorted array.

The file will be at least in two blocks b_1, b_2.

That is, disjointness is not satisfied

Choosing Partitions

- Choosing (or defining) partitions seems easy, but is easy to get wrong
- Consider the "*q: order of file F*"

b_1 = sorted in ascending order

b_2 = sorted in descending order

b_3 = arbitrary order

Solution:

Each characteristic should address just one property

q_1: File F sorted ascending

- **$q_1.b_1$ = true**
- **$q_1.b_2$ = false**

q_2: File F sorted descending

- **$q_2.b_1$ = true**
- **$q_2.b_2$ = false**

Example

Derive input space partitioning test inputs for the GenericStack class with the following method signatures:

- **public GenericStack ();**
- **public void push (Object X);**
- **public Object pop ();**
- **public boolean isEmpty ();**

Assume the usual semantics for the GenericStack.

Try to keep your partitioning simple and choose a small number of partitions and blocks.

Example

- public GenericStack ();
- public void push (Object X);
- public Object pop ();
- public boolean isEmpty ();

- Note that there are four testable units here
  - (the constructor and the three methods),
- But that there is substantial overlap between the characteristics relevant for each one.
- For the three methods, the implicit parameter(stack) is the state of the GenericStack.
- The only explicit input is the Object x parameter in Push().
- The constructor has neither inputs nor implicit parameters.

Example

- public GenericStack ();
- public void push (Object X);
- public Object pop ();
- public boolean isEmpty ();

Typical characteristics for the implicit state are

- q_1: Whether the stack is empty.
  - q_1b_1: true (Value stack = [])
  - q_1b_2: false (Values stack = [“cat”], [“cat”, “hat”])
- q_2: The size of the stack.
  - q_2b_1: 0 (Value stack = [])
  - q_2b_2: 1 (Possible values stack = [“cat”], [null])
  - q_2b_3: more than 1 (Possible values stack = [“cat”, “hat”], [“cat”, null], [“cat”, “hat”, “ox”])
- q_3: Whether the stack contains null entries
  - q_3b_1: true (Possible values stack = [null], [null, “cat”, null])
  - q_3b_2: false (Possible values stack = [“cat”, “hat”], [“cat”, “hat”, “ox”])

Example

- public GenericStack ();
- public void push (Object X);
- public Object pop ();
- public boolean isEmpty ();

A typical characteristic for Object x is

- q_4: Whether x is null.
  - q_4b_1: true (Value x = null)
  - q_4b_2: false (Possible values x = “cat”, “hat”, “”)

There are also characteristics that involves the combination of Object x and the stack state. One is:

- q_5: Does Object x appear in the stack?
  - q_5b_1: true (Possible values:
    (null, [null, “cat”, null]), (“cat”, [“cat”, “hat”]))
  - q_5b_2: false (Possible values: (null, [“cat”]), (“cat”, [“hat”, “ox”]))

Input Domain Modeling

- Interface-based approach
  - Develops characteristics directly from individual input parameters
  - Simplest application
  - Can be partially automated in some situations

- Functionality-based approach
  - Develops characteristics from a behavioral view of the program under test
  - Harder to develop—requires more design effort
  - May result in better tests, or fewer tests that are as effective

Input Domain Model (IDM)

Functionality-Based Approach

- Identify characteristics that correspond to the intended functionality
- Requires more design effort from tester
- Can incorporate domain and semantic knowledge
- Can use relationships among parameters
- Modeling can be based on requirements, not implementation
- The same parameter may appear in multiple characteristics, so it's harder to translate values to test cases

Interface-Based IDM – TriTyp

- TriTyp Program had one testable function and three integer inputs

First Characterization of TriTyp's Inputs

Partition	b_1	b_2	b_3
q_1 = "Relation of Side 1 to 0"	greater than 0	equal to 0	less than 0
q_2 = "Relation of Side 2 to 0"	greater than 0	equal to 0	less than 0
q_3 = "Relation of Side 3 to 0"	greater than 0	equal to 0	less than 0

- A maximum of 3*3*3 = 27 tests (e.g. q1 (7,?,?), (0,?,?), (-3,?,?))
- Some triangles are valid, some are invalid
- Refining the characterization can lead to more tests …

Functionality-Based IDM – TriTyp

- First two characterizations are based on syntax –parameters and their type
- A semantic level characterization could use the fact that the three integers represent a triangle

Geometric Characterization of TriTyp's Inputs

Partition	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	scalene	isosceles	equilateral	invalid

- Oops … something's fishy … equilateral is also isosceles !
- We need to refine the example to make partitions valid

Correct Geometric Characterization of TriTyp's Inputs

Partition	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	scalene	isosceles, not equilateral	equilateral	invalid

- Input S1, S2, S3
- Output= tritype= S,I,e,invalid

Functionality-Based IDM – TriTyp (*cont*)

- Values for this partitioning can be chosen as

Possible values for geometric partition q_1

Partition	b_1	b_2	b_3	b_4
q_1 = "Geometric Classification"	scalene	isosceles, not equilateral	equilateral	invalid
Triangle	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)

Functionality-Based IDM – TriTyp (*cont*)

- A different approach would be to break the geometric characterization into four separate characteristics

Four Characteristics for TriTyp

Partition	b_1	b_2
q_1 = "Scalene"	True	False
q_2 = "Isosceles"	True	False
q_3 = "Equilateral"	True	False
q_4 = "Valid"	True	False

Refinement of Functionality-Based *IDM—triang*()

```
public enum Triangle { Scalene, Isosceles, Equilateral, Invalid }
public static Triangle triang (int Side1, int Side2, int Side3)
// Side1, Side2, and Side3 represent the lengths of the sides of a triangle
// Returns the appropriate enum value
```

Relation among Sides Characterization of *triang*()'s Inputs

p = Side1 ≠ Side2 ≠ Side3

q = Side1 = Side2 ≠ Side3 | Side2 = Side3 ≠ Side1 | Side1 = Side3 ≠ Side2

r = Side1 = Side2 = Side3

s = Side1 + Side2 > Side3 | Side2 + Side3 > Side1 | Side1 + Side3 > Side2

t = Side1 > 0 & Side2 > 0 & Side3 > 0

Characteristic	b_1	b_2	b_3	b_4
q_{12} = "The Relation among Sides "	p & s & t	q & s & t	r & s & t	¬ s | ¬ t

Refinement of Functionality-Based *IDM—triang*()

- Values for this partitioning can be chosen as

Possible values for geometric partition q_{12}

Characteristic	b_1	b_2	b_3	b_4
q_{12} = "Relation among Sides"	(4, 5, 6)	(3, 3, 4)	(3, 3, 3)	(3, 4, 8)

- What are the expected outputs?

Refinement of Functionality-Based *IDM—triang*()

- A different approach would be to break the Relation among Sides characterization into four separate characteristics

p = Side1 ≠ Side2 ≠ Side3

q = Side1 = Side2 ≠ Side3 | Side2 = Side3 ≠ Side1 | Side1 = Side3 ≠ Side2

r = Side1 = Side2 = Side3

s = Side1 + Side2 > Side3 | Side2 + Side3 > Side1 | Side1 + Side3 > Side2

t = Side1 > 0 & Side2 > 0 & Side3 > 0

Four Characteristics for *triang*()

Characteristic	b_1	b_2
q_{13} = "p & s & t"	True	False
q_{14} = "q & s & t"	True	False
q_{15} = "r & s & t"	True	False
q_{16} = "s & t"	True	False

Choosing Combinations of Values

- Once characteristics and partitions(blocks) are defined, the next step is to choose test values
- We use criteria – to choose effective subsets
- The most obvious criterion is to choose all combinations …

All Combinations (ACoC) : One value from every block for every characteristic must be used with one value from every block for every other characteristic.

Abstract Example

For example, if we have three partitions with blocks

q1:[A, B], q2:[1, 2, 3], q3:[x, y],

then ACoC will need the following twelve tests:

(A, 1, x)	(B, 1, x)
(A, 1, y)	(B, 1, y)
(A, 2, x)	(B, 2, x)
(A, 2, y)	(B, 2, y)
(A, 3, x)	(B, 3, x)
(A, 3, y)	(B, 3, y)

- (A,1,x) means
  - a value from block A of characteristic q1,
  - a value from block 1 of characteristic q2,
  - and a value from block x of characteristic q3

ISP Criteria – Each Choice

- One criterion comes from the idea that we should try at least one value from each block

Each Choice (ECC) : One value from each block for each characteristic must be used in at least one test case.

q1:[A,B] q2:[1,2,3] q3:[x,y]

For Abstract Example:

(A,1,x)

(B,2,y)

(A,3,x)

ISP Criteria – Pair-Wise

- Each choice yields few tests – cheap but perhaps ineffective
- Another approach asks values to be combined with other values

Pair-Wise (PWC) : A value from each block for each characteristic must be combined with a value from every block for each other characteristic.

Abstract Example(Cont.)

- Given the above example of three partitions with blocks then PWC will need tests to cover the following 16 combinations:

(A, 1)	(B, 1)	(1, x)
(A, 2)	(B, 2)	(1, y)
(A, 3)	(B, 3)	(2, x)
(A, x)	(B, x)	(2, y)
(A, y)	(B, y)	(3, x)
		(3, y)

q1:[A,B] q2:[1,2,3] q3:[x,y]

- PWC allows the same test case to cover more than one unique pair of values. So the above combinations can be combined in several ways, including:

(A, 1, x)	(B, 1, y)
(A, 2, x)	(B, 2, y)
(A, 3, x)	(B, 3, y)
(A, -, y)	(B, -, x)

ISP Criteria – Base Choice

- Testers sometimes recognize that certain values are important
- This uses domain knowledge of the program

Base Choice (BCC) : A base choice block is chosen for each characteristic, and a base test is formed by using the base choice for each characteristic. Subsequent tests are chosen by holding all but one base choice constant and using each non-base choice in each other characteristic.

Abstract Example(Cont.)

q1:[A,B] q2:[1,2,3] q3:[x,y]

- Given the above example of three partitions suppose
  - base choice blocks are ‘A’, ‘1’ and ‘x’.
  - Then the base choice test is (A, 1, x), and
  - the following additional tests would be needed:

(A, 1, y)

(A, 2, x)

(A, 3, x)

(B, 1, x)

Example

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          else return true if element is in the list, false otherwise
```

Characteristic	Block 1	Block 2	Block 3	Block 4
A:				

Example

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          else return true if element is in the list, false otherwise
```

Characteristic	Block 1	Block 2	Block 3	Block 4
A : length and contents				

Example

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          else return true if element is in the list, false otherwise
```

Characteristic	Block 1	Block 2	Block 3	Block 4
A : length and contents	One element	More than one, unsorted	More than one, sorted	More than one, all identical

Is this characteristic OK?

Example

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          else return true if element is in the list, false otherwise
```

Characteristic	Block 1	Block 2	Block 3	Block 4
A : length and contents	Less than two elements	More than one, different and unsorted	More than one, different and sorted	More than one, all identical

What does different mean?

All elements are different or

Some of the elements are different?

If we assume All elements are different then what about completeness?

So by different we mean: Some of the elements are different?

Example

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          else return true if element is in the list, false otherwise
```

Characteristic	Block 1	Block 2	Block 3	Block 4
B :				

Example

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          else return true if element is in the list, false otherwise
```

Characteristic	Block 1	Block 2	Block 3	Block 4
B : match				

Example

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          else return true if element is in the list, false otherwise
```

Characteristic	Block 1	Block 2	Block 3	Block 4
B : match	element not found	element found once	element found more than once	

Is this characteristic OK?

Example

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          else return true if element is in the list, false otherwise
```

Characteristic	Block 1	Block 2	Block 3	Block 4
B : element and list	element is not in the list	element is in the list only once	element is in the list more than once	

Example

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          else return true if element is in the list, false otherwise
```

Characteristic	Block 1	Block 2	Block 3	Block 4
A : length and contents	Less than two elements	More than one, different and unsorted	More than one, different and sorted	More than one, all identical
B : element and list	element is not in the list	element is in the list only once	element is in the list more than once	

Example Handling Constraints

```
public boolean findElement (List list, Object element)
// Effects: if list or element is null throw NullPointerException
//          else return true if element is in the list, false otherwise
```

Characteristic	Block 1	Block 2	Block 3	Block 4
A : length and contents	Less than two elements	More than one, different and unsorted	More than one, different and sorted	More than one, all identical
B : element and list	element is not in the list	element is in the list only once	element is in the list more than once	
Invalid combinations : (**A1**, **B3**), (**A4**, **B2**)				

element cannot be in a less-than-two-elements list more than once

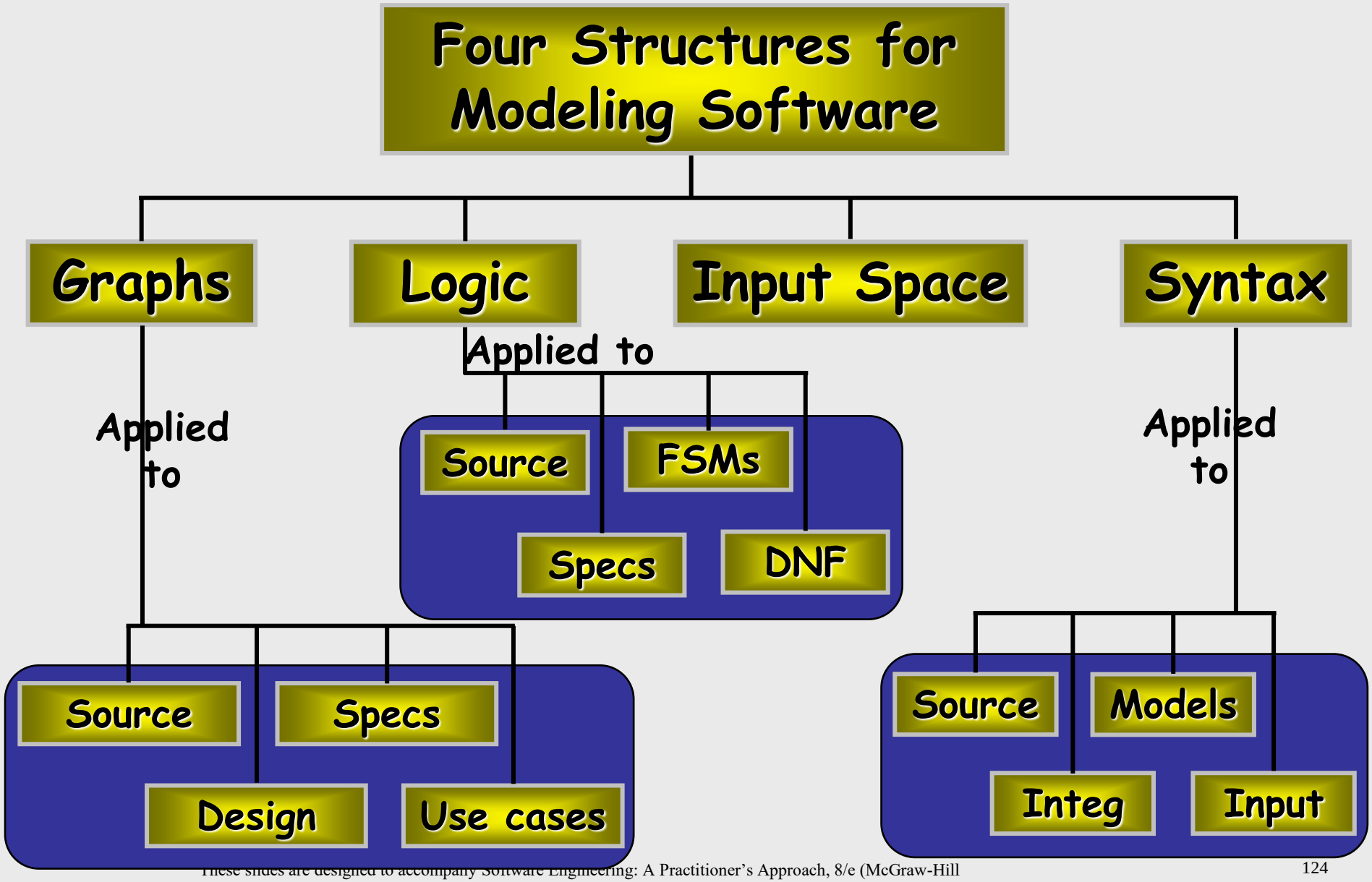
If the list only has one element, but it appears multiple times, it cannot be in the list just once

Input Space Partitioning Summary

- Fairly easy to apply, even with no automation
- Convenient ways to add more or less testing
- Applicable to all levels of testing – unit, class, integration, system, etc.
- Based only on the input space of the program, not the implementation

Simple, straightforward, effective, and widely used

Model Driven Testing: Coverage Overview



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

Chapter 24

Testing Object-Oriented Applications

Slide Set to accompany

Software Engineering: A Practitioner's Approach, 8/e

by Roger S. Pressman and Bruce R.Maxim

OO Testing

- To adequately test OO systems, three things must be done:
  - the definition of testing must be broadened to include error discovery techniques applied to object-oriented analysis and design models
  - the strategy for unit and integration testing must change significantly, and
  - the design of test cases must account for the unique characteristics of OO software.

OO Testing Strategies-Unit testing

- the concept of the unit changes
- the smallest testable unit is the encapsulated class
- a single operation can no longer be tested in isolation (the conventional view of unit testing) but rather, as part of a class

OO Testing Strategies- Integration Testing

- *Thread-based testing*
  - integrates the set of classes required to respond to one input or event for the system
- *Use-based testing*
  - begins the construction of the system by
    - testing those classes (called *independent classes*) that use very few (if any) of server classes.
    - After the independent classes are tested, the next layer of classes, called *dependent classes, that use the independent classes are tested.*
- *Cluster testing*:
  - A cluster of collaborating classes (determined by examining the CRC and object-relationship model) is exercised by designing test cases that attempt to uncover errors in the collaborations.

OO Testing Methods

- Class Testing and the
- Class Hierarchy
  - Inheritance does not obviate(prevent) the need for thorough testing of all derived classes.
  - In fact, it can actually complicate the testing process.

End of Chapters 22-23-24

Metrics for Testing 1

Although much has been written on software metrics for testing (e.g., [HET93]), the majority of metrics proposed focus on the process of testing, not the technical characteristics of the tests themselves. In general, testers must rely on analysis, design, and code metrics to guide them in the design and execution of test cases.

- Architectural design metrics provide information on the ease or difficulty associated with integration testing and the need for specialized testing software (e.g., stubs and drivers).
- Modules with high cyclomatic complexity are more likely to be error prone than modules whose cyclomatic complexity is lower.

Metrics for Testing 2

Software science assigns quantitative laws to the development of computer software, using a set of primitive measures that may be derived after code is generated or estimated once design is complete. These follow:

n_1 = the number of distinct operators that appear in a program.
n_2 = the number of distinct operands that appear in a program.
N_1 = the total number of operator occurrences.
N_2 = the total number of operand occurrences.

Metrics for Testing 3

Halstead shows that length N can be estimated

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

and program volume may be defined

$$V = N \log_2 (n_1 + n_2)$$

V will vary with programming language and represents the volume of information (in bits) required to specify a program.

Theoretically, a minimum volume must exist for a particular algorithm.

Halstead defines a volume ratio L as the ratio of volume of the most compact form of a program to the volume of the actual program.

In terms of primitive measures, the volume ratio may be expressed as

$$L = 2/n_1 \times n_2/N_2$$

In actuality, L must always be less than 1.

Metrics for Testing 4

- Testing effort can also be estimated using metrics derived from Halstead measures
- Using the definitions for program volume, V, and program level, PL, software science effort, e, can be computed as

$$PL = 1/[(n_1/2) \times (N_2/n_2)]$$

$$e = V/PL$$

- The percentage of overall testing effort to be allocated to a module k can be estimated using the following relationship:

$$\text{percentage of testing effort } (k) = e(k)/\Sigma\, e(i)$$

where $e(k)$ is computed for module k using Equations $e = V/PL$ and the summation in the denominator of Equation is the sum of software science effort across all modules of the system.

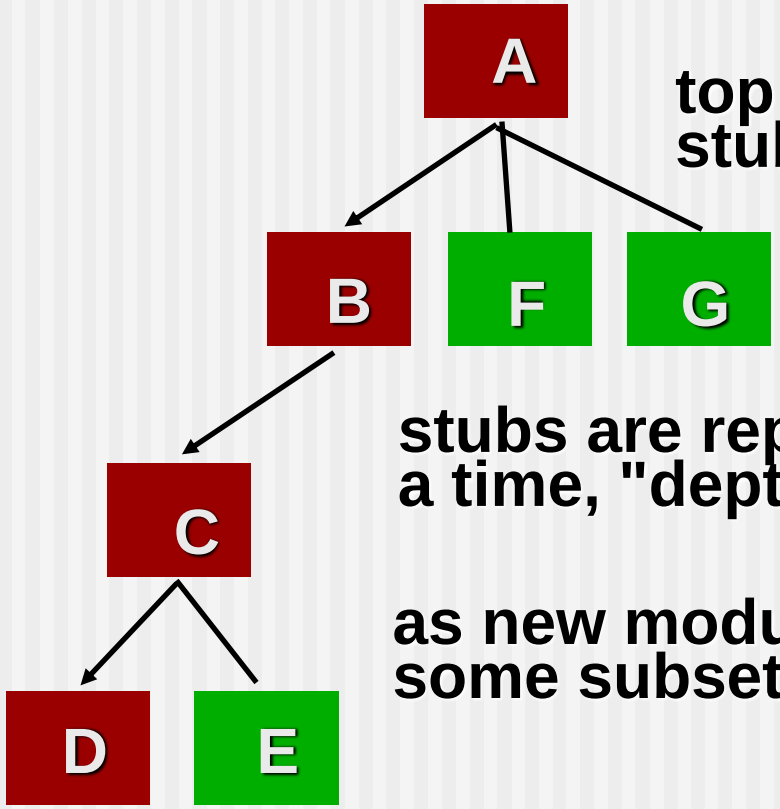
Metrics for Testing 5

- Binder [BIN94] suggests a broad array of design metrics that have a direct influence on the "testability" of an OO system.
  - Lack of cohesion in methods (LCOM).
  - Percent public and protected (PAP).
  - Public access to data members (PAD).
  - Number of root classes (NOR).
  - Fan-in (FIN).
  - Number of children (NOC) and depth of the inheritance tree (DIT).

Maintenance Metrics

- IEEE Std. 982.1-1988 [IEE94] suggests a *software maturity index* (SMI) that provides an indication of the stability of a software product (based on changes that occur for each release of the product). The following information is determined:
  - M_T = the number of modules in the current release
  - F_c = the number of modules in the current release that have been changed
  - F_a = the number of modules in the current release that have been added
  - F_d = the number of modules from the preceding release that were deleted in the current release
- The software maturity index is computed in the following manner:
  - SMI = $[M_T - (F_a + F_c + F_d)]/M_T$
- As SMI approaches 1.0, the product begins to stabilize.

Top Down Integration

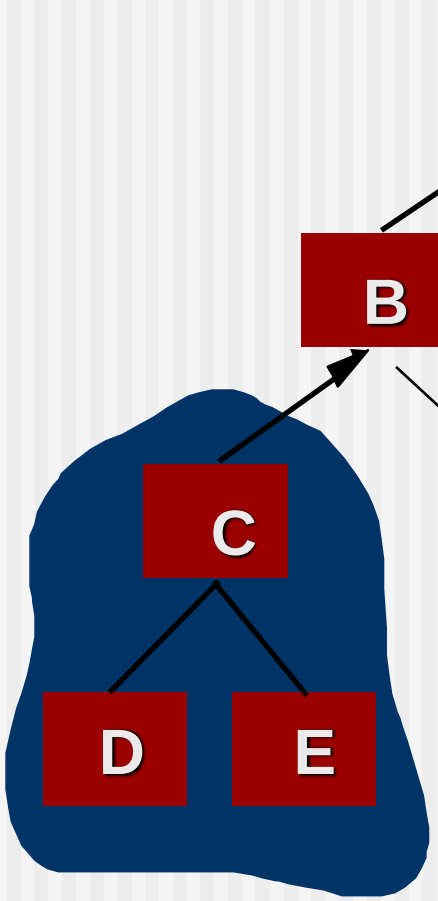


top module is tested with stubs

stubs are replaced one at a time, "depth first"

as new modules are integrated, some subset of tests is re-run

Bottom-Up Integration

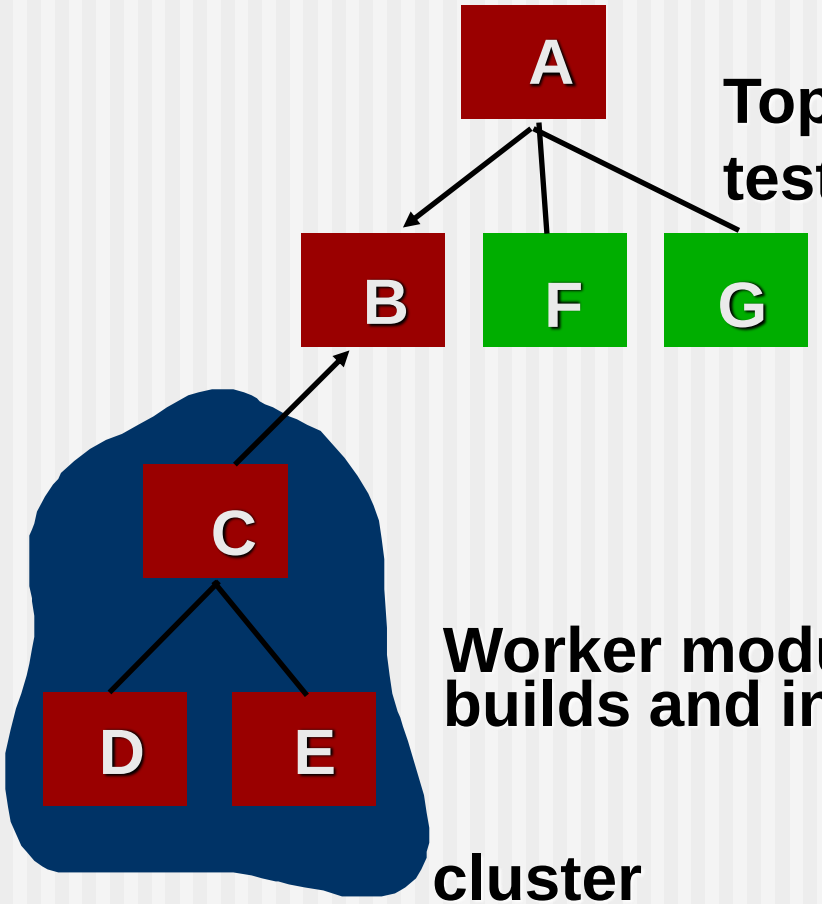


drivers are replaced one at a time, "depth first"

worker modules are grouped int builds and integrated

cluster

Sandwich Testing



Top modules are tested with stubs

Worker modules are grouped into builds and integrated

cluster

Smoke Testing

- A common approach for creating “daily builds” for product software
- Smoke testing steps:
  - Software components that have been translated into code are integrated into a “build.”
    - A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
  - A series of tests is designed to expose errors that will keep the build from properly performing its function.
    - The intent should be to uncover “show stopper” errors that have the highest likelihood of throwing the software project behind schedule.
  - The build is integrated with other builds and the entire product (in its current form) is smoke tested daily.
    - The integration approach may be top down or bottom up.

General Testing Criteria

- Interface integrity – internal and external module interfaces are tested as each module or cluster is added to the software
- Functional validity – test to uncover functional defects in the software
- Information content – test for errors in local or global data structures
- Performance – verify specified performance bounds are tested

Object-Oriented Testing

- begins by evaluating the correctness and consistency of the analysis and design models
- testing strategy changes
  - the concept of the ‘unit’ broadens due to encapsulation
  - integration focuses on classes and their execution across a ‘thread’ or in the context of a usage scenario
  - validation uses conventional black box methods
- test case design draws on conventional methods, but also encompasses special features

Broadening the View of "Testing"

It can be argued that the review of OO analysis and design models is especially useful because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code level. Therefore, a problem in the definition of class attributes that is uncovered during analysis will circumvent side effects that might occur if the problem were not discovered until design or code (or even the next iteration of analysis).

Testing the CRC Model

1. Revisit the CRC model and the object-relationship model.
2. Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator's definition.
3. Invert the connection to ensure that each collaborator that is asked for service is receiving requests from a reasonable source.
4. Using the inverted connections examined in step 3, determine whether other classes might be required or whether responsibilities are properly grouped among the classes.
5. Determine whether widely requested responsibilities might be combined into a single responsibility.
6. Steps 1 to 5 are applied iteratively to each class and through each evolution of the analysis model.

OO Testing Strategy

- class testing is the equivalent of unit testing
  - operations within the class are tested
  - the state behavior of the class is examined
- integration applied three different strategies
  - thread-based testing—integrates the set of classes required to respond to one input or event
  - use-based testing—integrates the set of classes required to respond to one use case
  - cluster testing—integrates the set of classes required to demonstrate one collaboration

WebApp Testing - I

- The content model for the WebApp is reviewed to uncover errors.
- The interface model is reviewed to ensure that all use cases can be accommodated.
- The design model for the WebApp is reviewed to uncover navigation errors.
- The user interface is tested to uncover errors in presentation and/or navigation mechanics.
- Each functional component is unit tested.

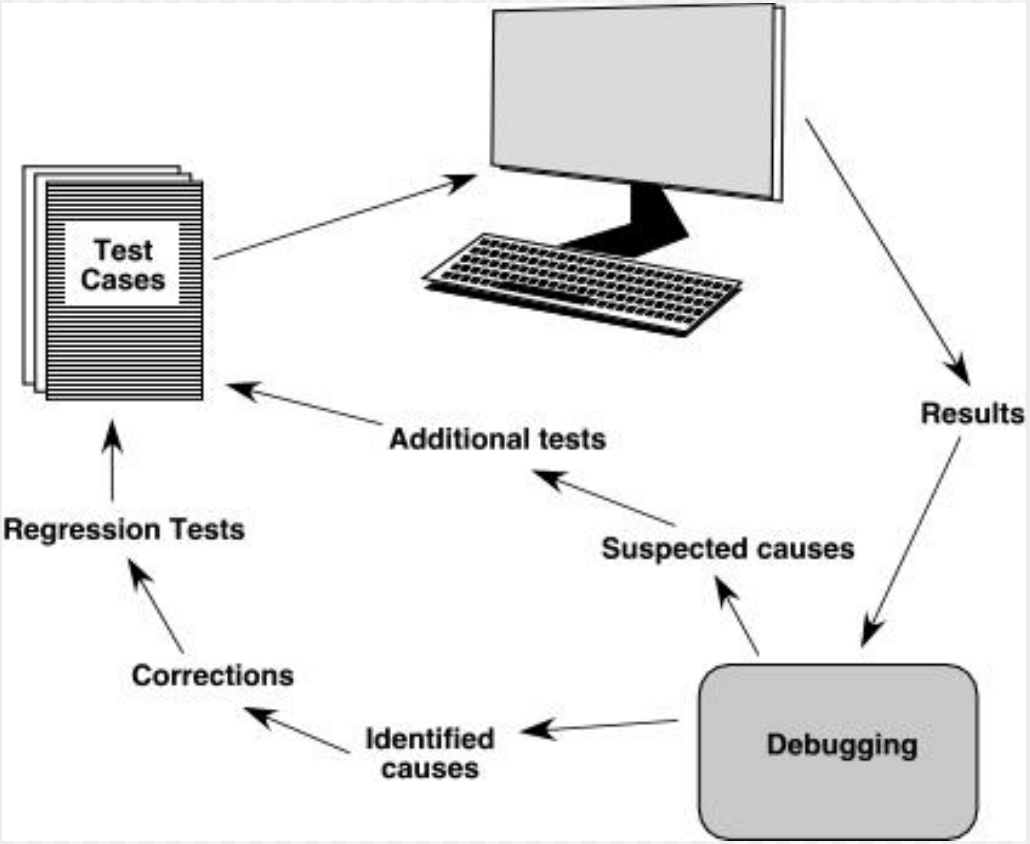
WebApp Testing - II

- Navigation throughout the architecture is tested.
- The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.
- Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.
- Performance tests are conducted.
- The WebApp is tested by a controlled and monitored population of end-users. The results of their interaction with the system are evaluated for content and navigation errors, usability concerns, compatibility concerns, and WebApp reliability and performance.

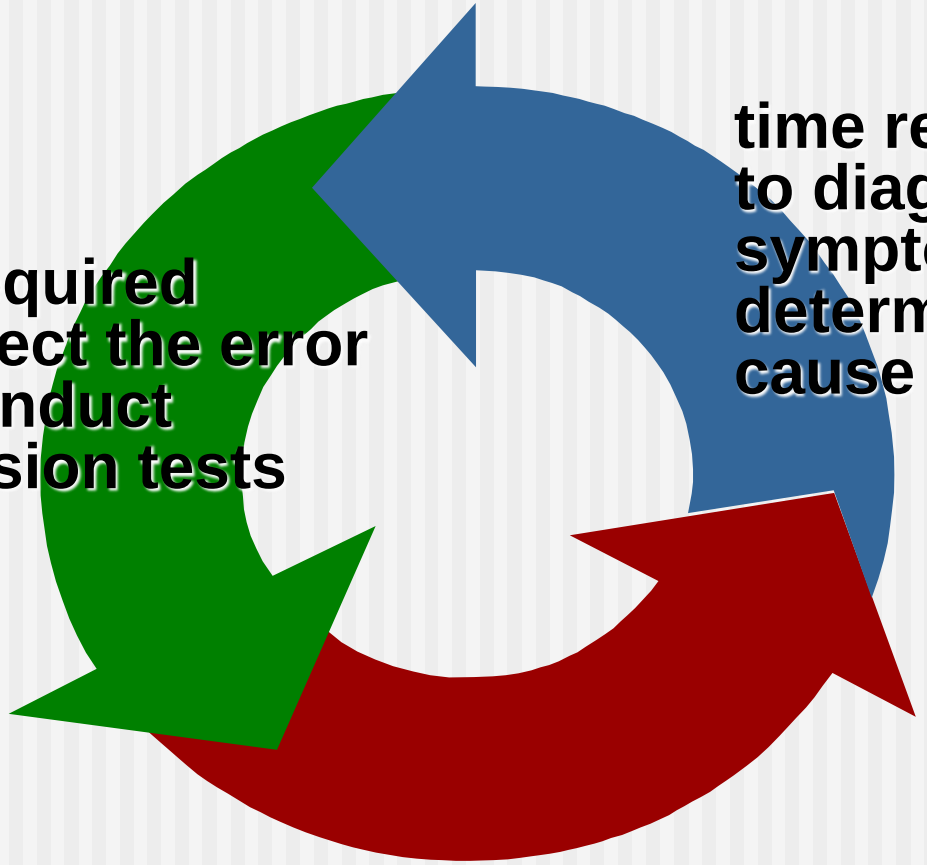
MobileApp Testing

- User experience testing – ensuring app meets stakeholder usability and accessibility expectations
- Device compatibility testing – testing on multiple devices
- Performance testing – testing non-functional requirements
- Connectivity testing – testing ability of app to connect reliably
- Security testing – ensuring app meets stakeholder security expectations
- Testing-in-the-wild – testing app on user devices in actual user environments
- Certification testing – app meets the distribution standards

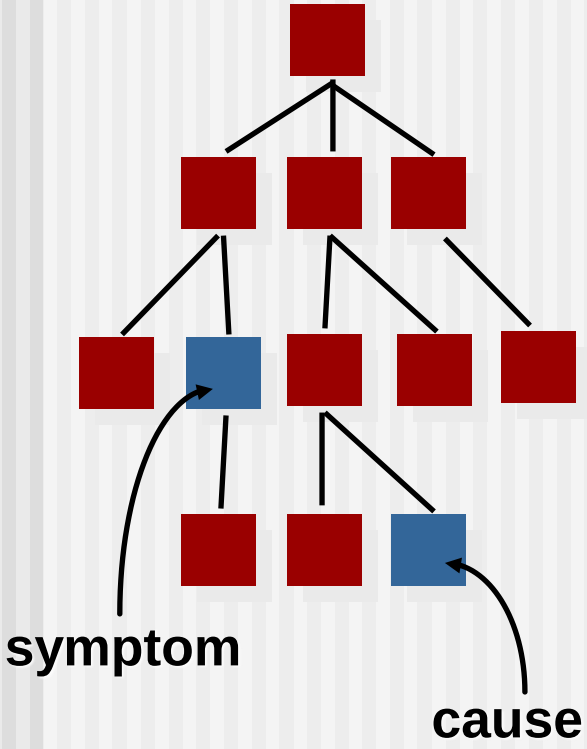
The Debugging Process



Debugging Effort

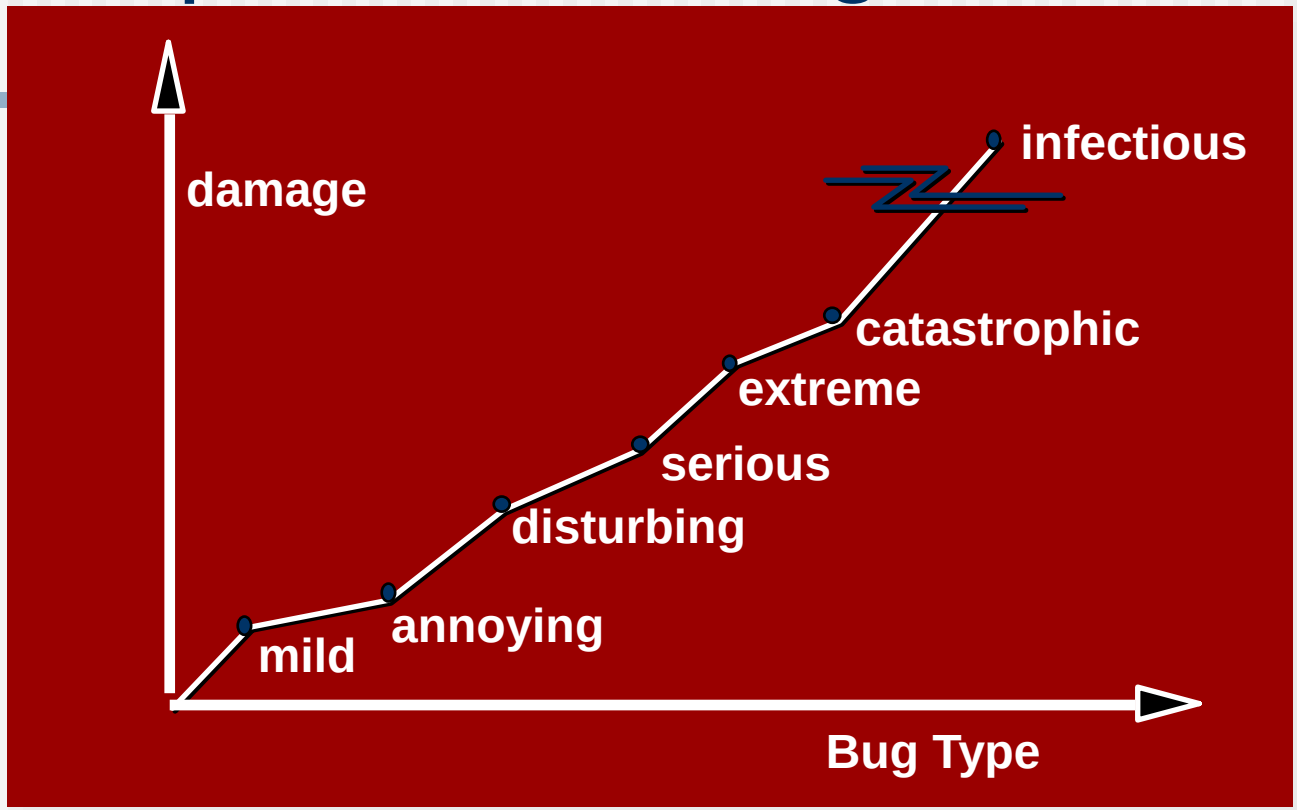


Symptoms & Causes



- symptom and cause may be geographically separated
- symptom may disappear when another problem is fixed
- cause may be due to a combination of non-errors
- cause may be due to a system or compiler error
- cause may be due to assumptions that everyone believes
- symptom may be intermittent

Consequences of Bugs



Bug Categories: **function-related bugs, system-related bugs, data bugs, coding bugs, design bugs, documentation bugs, standards violations, etc.**

Debugging Techniques

- **brute force / testing**
- **backtracking**
- **induction**
- **deduction**

Correcting the Error

- *Is the cause of the bug reproduced in another part of the program?* In many situations, a program defect is caused by an erroneous pattern of logic that may be reproduced elsewhere.
- *What "next bug" might be introduced by the fix I'm about to make?* Before the correction is made, the source code (or, better, the design) should be evaluated to assess coupling of logic and data structures.
- *What could we have done to prevent this bug in the first place?* This question is the first step toward establishing a statistical software quality assurance approach. If you correct the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs.

Final Thoughts

- *Think* -- before you act to correct
- Use tools to gain additional insight
- If you're at an impasse, get help from someone else
- Once you correct the bug, use regression testing to uncover any side effects

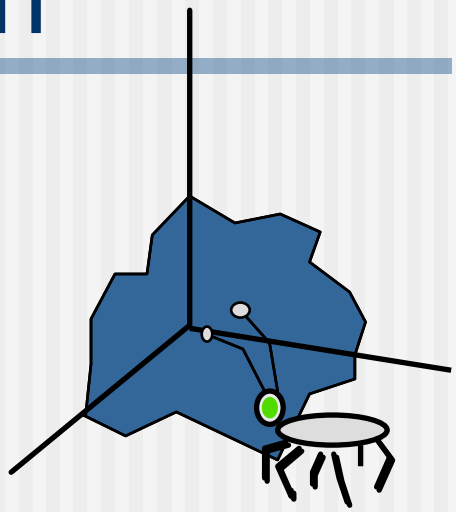
Internal and External Views

- 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

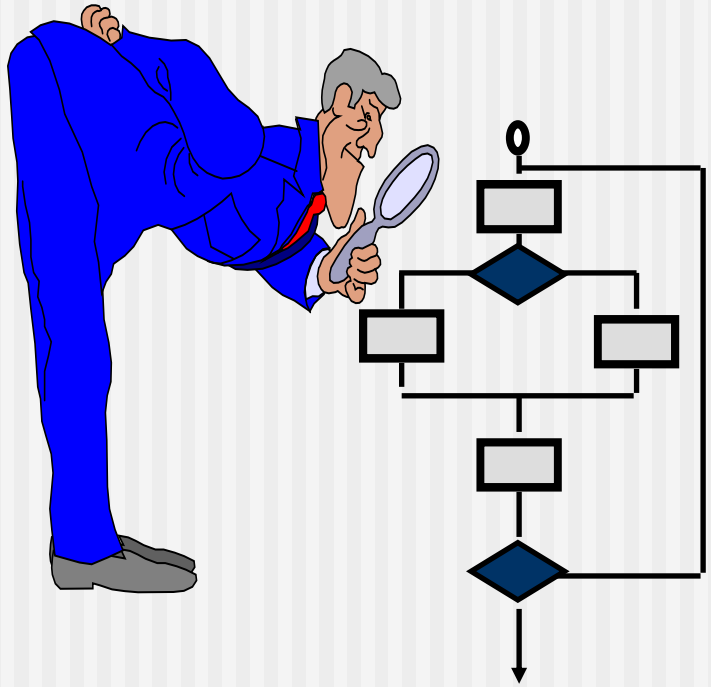


OBJECTIVE **to uncover errors**

CRITERIA **in a complete manner**

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

White-Box Testing



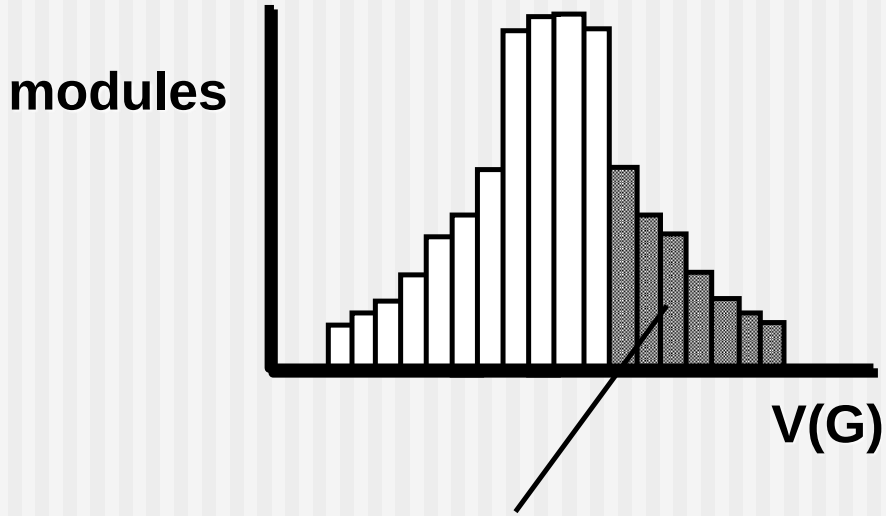
... our goal is to ensure that all statements and conditions have been executed at least once ...

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**

Cyclomatic Complexity

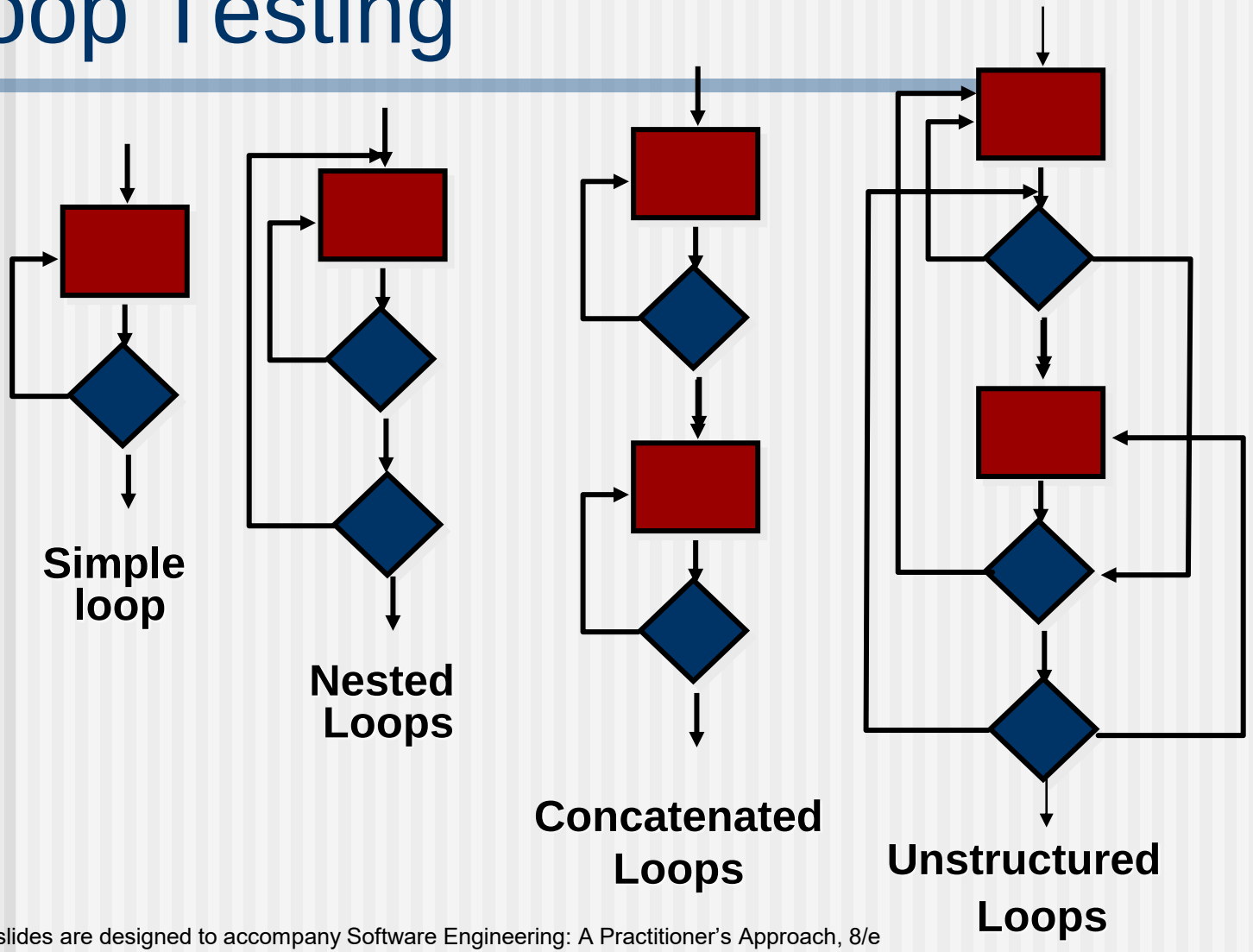
A number of industry studies have indicated that the higher V(G), the higher the probability or errors.



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

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.

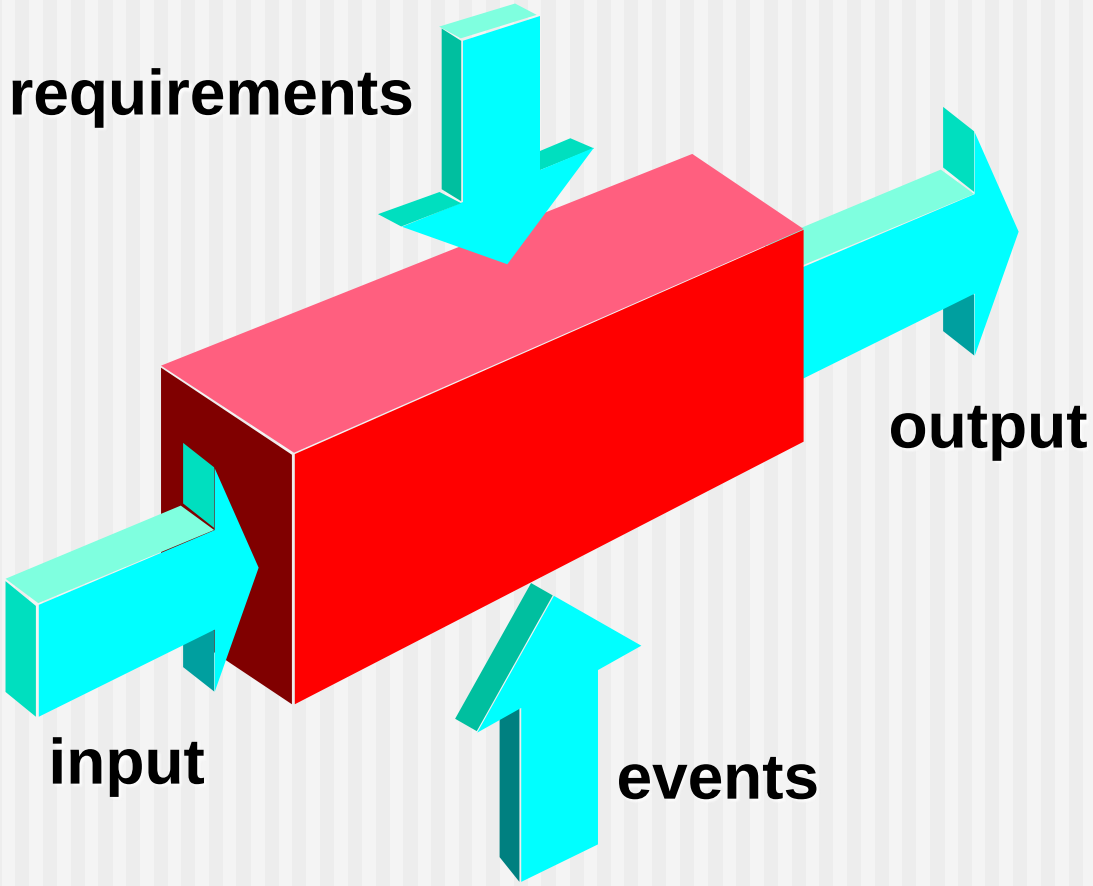
Move out one loop and set it up as in step 2, holding all other loops at typical values. Continue this step until the outermost loop has been tested.

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



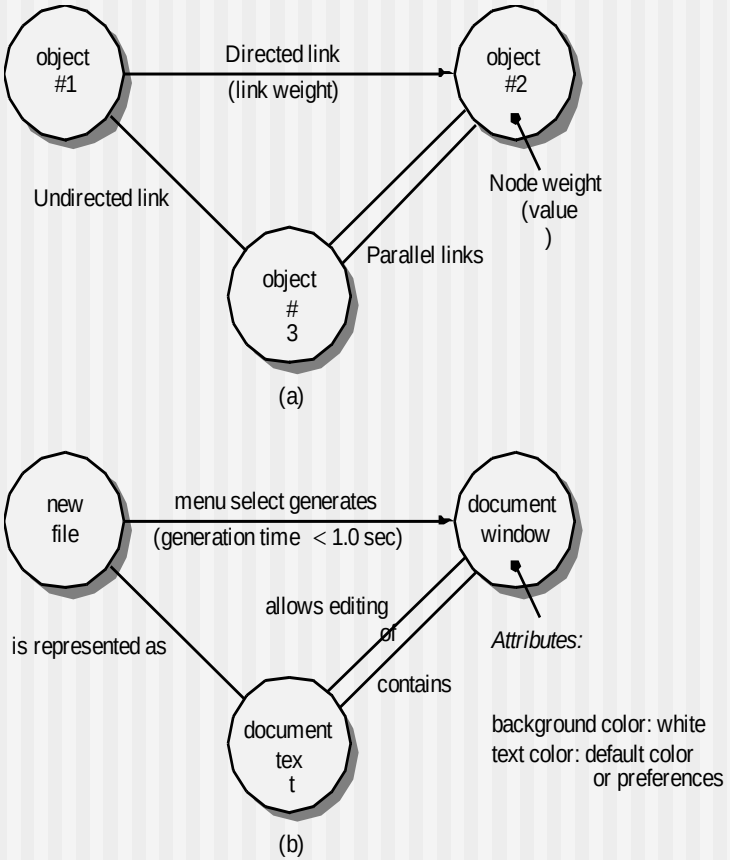
Black-Box Testing

- 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?

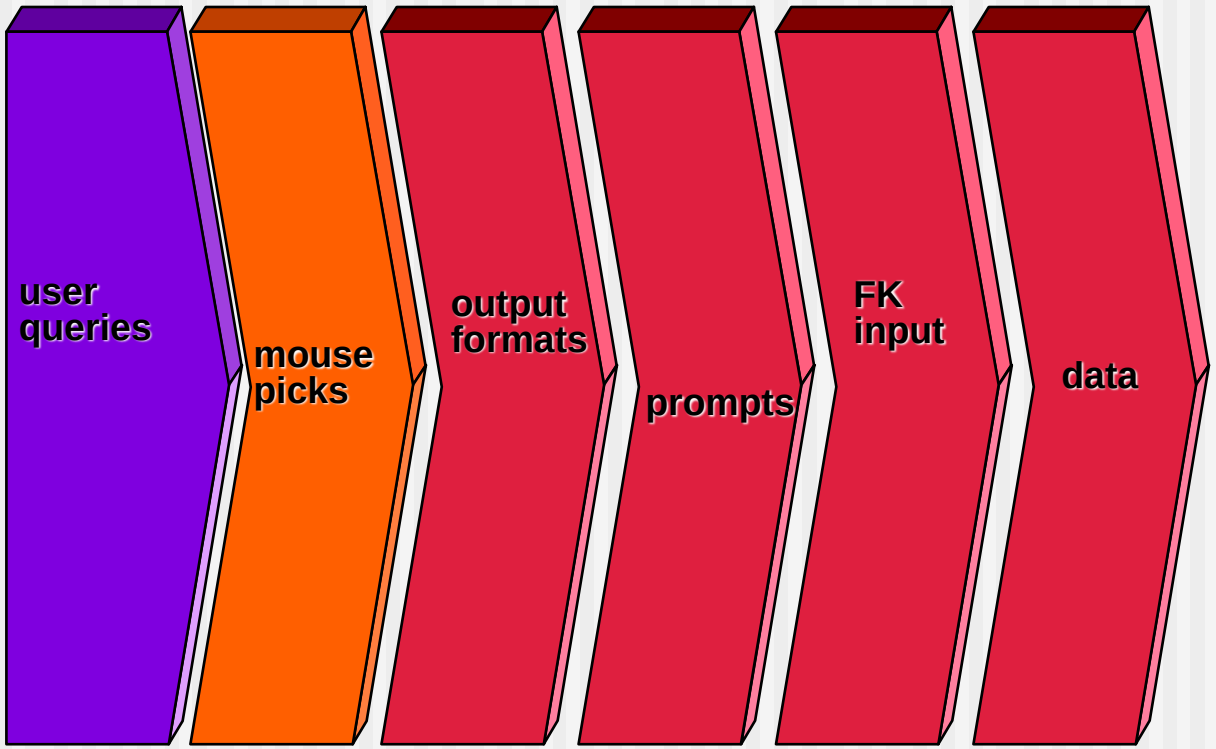
Graph-Based Methods

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

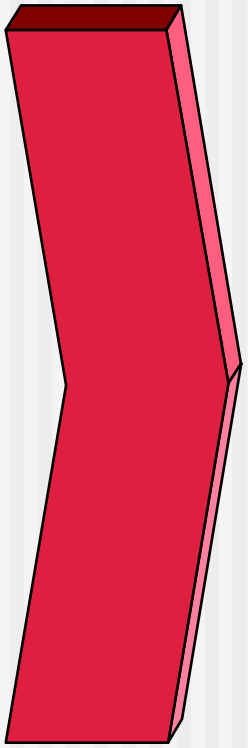
In this context, we consider the term "objects" in the broadest possible context. It encompasses data objects, traditional components (modules), and object-oriented elements of computer software.



Equivalence Partitioning



Sample Equivalence Classes



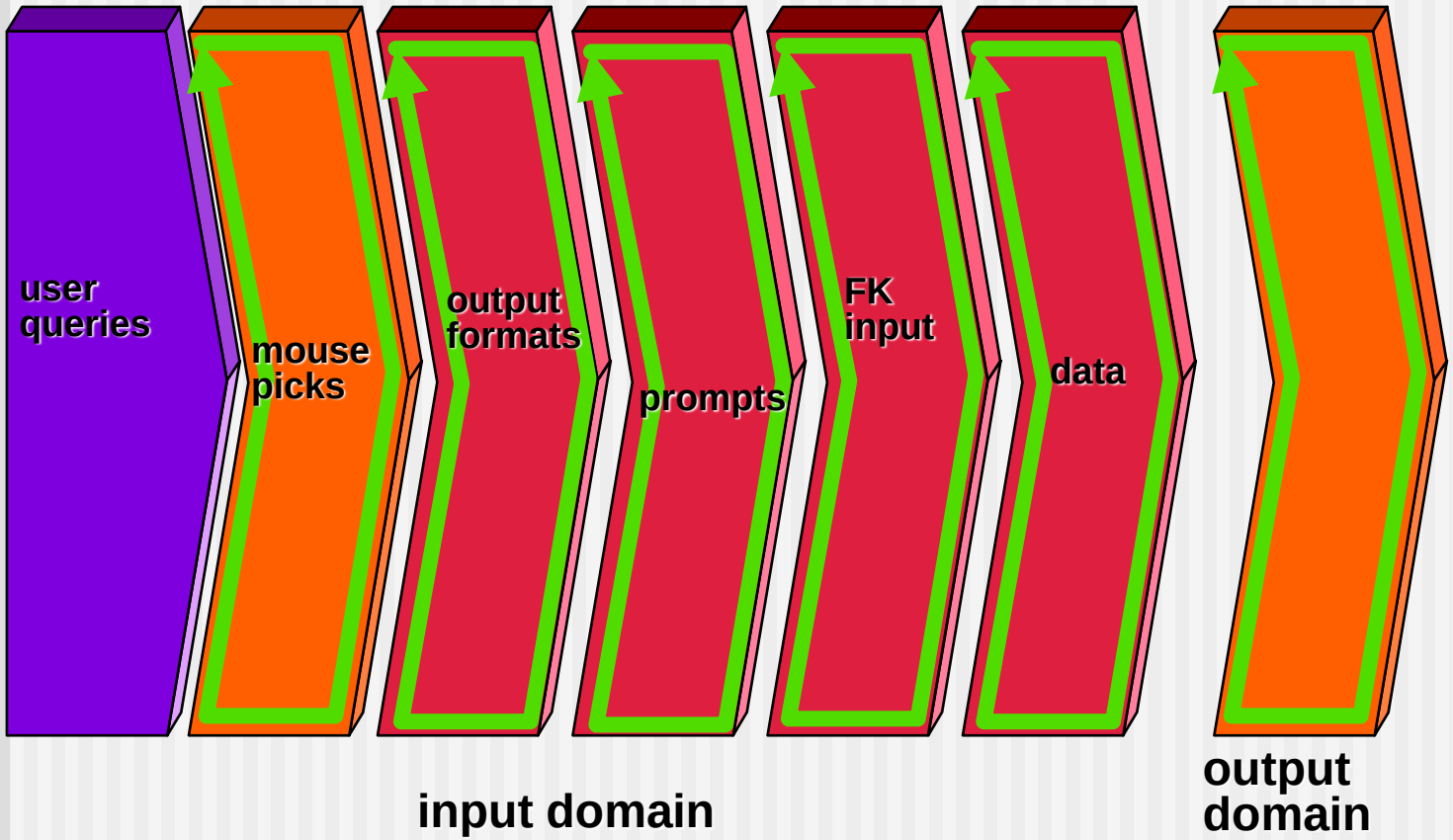
Valid data

- **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



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]

‘Testing’ OO Models

- The review of OO analysis and design models is especially useful because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code level
- Therefore, a problem in the definition of class attributes that is uncovered during analysis will circumvent side affects that might occur if the problem were not discovered until design or code (or even the next iteration of analysis).

Correctness of OO Models

- During analysis and design, semantic correctness can be asesssed based on the model's conformance to the real world problem domain.
- If the model accurately reflects the real world (to a level of detail that is appropriate to the stage of development at which the model is reviewed) then it is semantically correct.
- To determine whether the model does, in fact, reflect real world requirements, it should be presented to problem domain experts who will examine the class definitions and hierarchy for omissions and ambiguity.
- Class relationships (instance connections) are evaluated to determine whether they accurately reflect real-world object connections.

Class Model Consistency

- Revisit the CRC model and the object-relationship model.
- Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator's definition.
- Invert the connection to ensure that each collaborator that is asked for service is receiving requests from a reasonable source.
- Using the inverted connections examined in the preceding step, determine whether other classes might be required or whether responsibilities are properly grouped among the classes.
- Determine whether widely requested responsibilities might be combined into a single responsibility.

OO Testing Strategies

- Validation Testing
  - details of class connections disappear
  - draw upon use cases (Chapters 5 and 6) that are part of the requirements model
  - Conventional black-box testing methods (Chapter 18) can be used to drive validation tests

OOT Methods

Berard [Ber93] proposes the following approach:

1. **Each test case should be uniquely identified and should be explicitly associated with the class to be tested,**
2. **The purpose of the test should be stated,**
3. **A list of testing steps should be developed for each test and should contain [BER94]:**
   a. **a list of specified states for the object that is to be tested**
   b. **a list of messages and operations that will be exercised as a consequence of the test**
   c. **a list of exceptions that may occur as the object is tested**
   d. **a list of external conditions (i.e., changes in the environment external to the software that must exist in order to properly conduct the test)**
   e. **supplementary information that will aid in understanding or implementing the test.**

Testing Methods

- Fault-based testing
  - The tester looks for plausible faults (i.e., aspects of the implementation of the system that may result in defects). To determine whether these faults exist, test cases are designed to exercise the design or code.
- Class Testing and the Class Hierarchy
  - Inheritance does not obviate the need for thorough testing of all derived classes. In fact, it can actually complicate the testing process.
- Scenario-Based Test Design
  - Scenario-based testing concentrates on what the user does, not what the product does. This means capturing the tasks (via use-cases) that the user has to perform, then applying them and their variants as tests.

OOT Methods: Random Testing

- Random testing
  - identify operations applicable to a class
  - define constraints on their use
  - identify a minimum test sequence
    - an operation sequence that defines the minimum life history of the class (object)
  - generate a variety of random (but valid) test sequences
    - exercise other (more complex) class instance life histories

OOT Methods: Partition Testing

- Partition Testing
  - reduces the number of test cases required to test a class in much the same way as equivalence partitioning for conventional software
  - state-based partitioning
    - categorize and test operations based on their ability to change the state of a class
  - attribute-based partitioning
    - categorize and test operations based on the attributes that they use
  - category-based partitioning
    - categorize and test operations based on the generic function each performs

OOT Methods: Inter-Class Testing

- Inter-class testing
  - For each client class, use the list of class operators to generate a series of random test sequences. The operators will send messages to other server classes.
  - For each message that is generated, determine the collaborator class and the corresponding operator in the server object.
  - For each operator in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits.
  - For each of the messages, determine the next level of operators that are invoked and incorporate these into the test sequence

OOT Methods: Behavior Testing

The tests to be designed should achieve all state coverage [KIR94]. That is, the operation sequences should cause the Account class to make transition through all allowable states