

SOFTWARE ENGINEERING

Software Implementation and Testing

Agenda

1. Implementation/Programming Guidelines
2. Software Testing Concepts
3. Unit Testing
4. Module Integration and Testing Strategies
5. Testing Approaches
 1. White-Box Testing
 2. Black-Box Testing
6. Other Types of Testing

1. Implementation/Programming Guidelines ←
2. Software Testing Concepts
3. Unit Testing
4. Module Integration and Testing Strategies
5. Testing Approaches
 1. White-Box Testing
 2. Black-Box Testing
6. Other Types of Testing

Implementation/Programming Guidelines

❧11.1❧

Good Programming Practice

☞ Use of *consistent* and *meaningful* variable names

- “Meaningful” to future maintenance programmers
- “Consistent” to aid future maintenance programmers

Use of Consistent and Meaningful Variable Names

- ✎ A code artifact includes the variable names `freqAverage`, `frequencyMaximum`, `minFr`, `frqncyTotl`
- ✎ A maintenance programmer has to know if `freq`, `frequency`, `fr`, `frqncy` all refer to the same thing
 - If so, use the identical word, preferably `frequency`, perhaps `freq` or `frqncy`, but *not* `fr`
 - If not, use a different word (e.g., `rate`) for a different quantity

Consistent and Meaningful Variable Names

- ✎ We can use `frequencyAverage`, `frequencyMaximum`, `frequencyMinimum`, `frequencyTotal`
- ✎ We can also use `averageFrequency`, `maximumFrequency`, `minimumFrequency`, `totalFrequency`
- ✎ But all four names must come from the same set

Establish and use a fixed ordering for javadoc tags.

☞ In class and interface descriptions, use:

`@author` *your name*

`@version` *a version number or date*

☞ In method descriptions, use:

`@param` *p* *A description of parameter p.*

`@return` *A description of the value returned
(unless it's void).*

`@exception` *e* *Describe any thrown exception.*

Tags in doc comments II

Fully describe the signature of each method.

- ✎ The signature is what distinguishes one method from another
 - the signature includes the number, order, and types of the parameters
- ✎ Use a **@param** tag to describe each parameter
 - **@param** tags should be in the correct order
 - Don't mention the parameter *type*; javadoc does that
 - Use a **@return** tag to describe the result (unless it's **void**)

Use of Parameters

✂ There are almost no genuine constants

✂ One solution:

- Use `const` statements (C++), or
- Use `public static final` statements (Java)

✂ A better solution:

- Read the values of “constants” from a parameter file

Input and output conditions

Document preconditions, postconditions, and invariant conditions.

- ✂ A precondition is something that must be true beforehand in order to use your method
 - Example: **The piece must be moveable**
- ✂ A postcondition is something that your method makes true
 - Example: **The piece is not against an edge**
- ✂ An invariant is something that must *always* be true about an object
 - Example: **The piece is in a valid row and column**

Nested `if` Statements (contd)

- ✎ A combination of `if-if` and `if-else-if` statements is usually difficult to read
- ✎ Simplify: The `if-if` combination

```
if <condition1>  
    if <condition2>
```

is frequently equivalent to the single condition

```
if <condition1> && <condition2>
```

- ✎ Rule of thumb
 - `if` statements nested to a depth of greater than three should be avoided as poor programming practice

Programming Standards

- ☞ Standards can be both a blessing and a curse
- ☞ Modules of coincidental cohesion arise from rules like
 - “Every module will consist of between 35 and 50 executable statements”
- ☞ Better
 - “Programmers should consult their managers before constructing a module with fewer than 35 or more than 50 executable statements”

Examples of Good Programming Standards

- ∞ “Nesting of `if` statements should not exceed a depth of 3, except with prior approval from the team leader”
- ∞ “Modules should consist of between 35 and 50 statements, except with prior approval from the team leader”
- ∞ These programming standards reduce the complexity of the code and improve its readability and maintainability.

Cyclomatic Complexity

- ∞ Cyclomatic Complexity $V(G)$ is defined as the number of regions in the flow graph.

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

E: number of edges in flow graph

N: number of nodes in flow graph

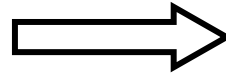
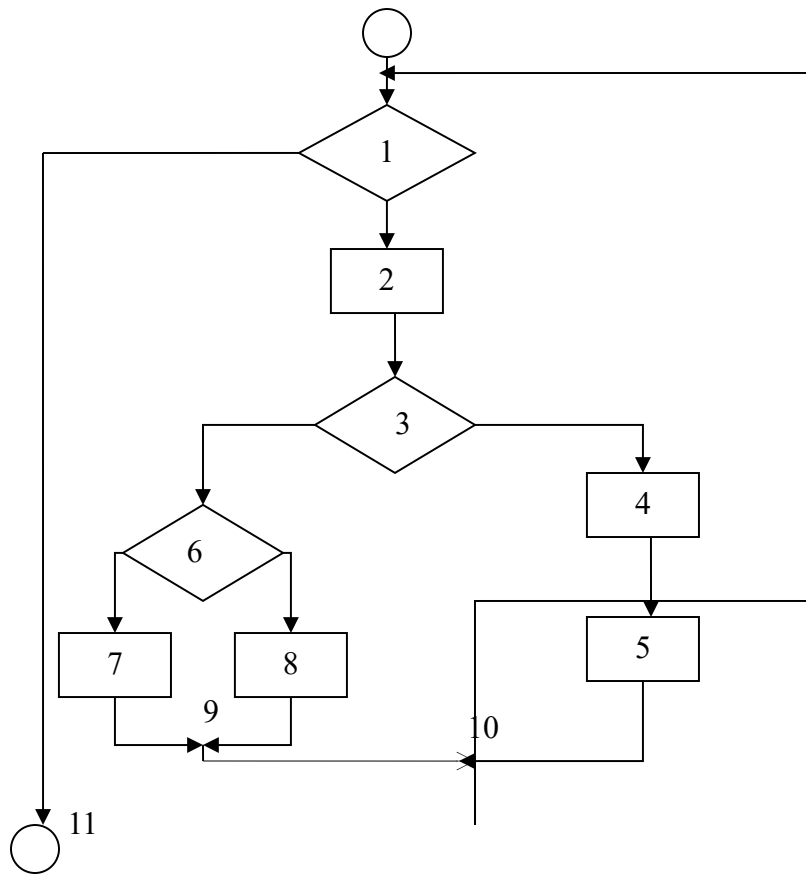
- ∞ **Another method:**

$$V(G) = P + 1$$

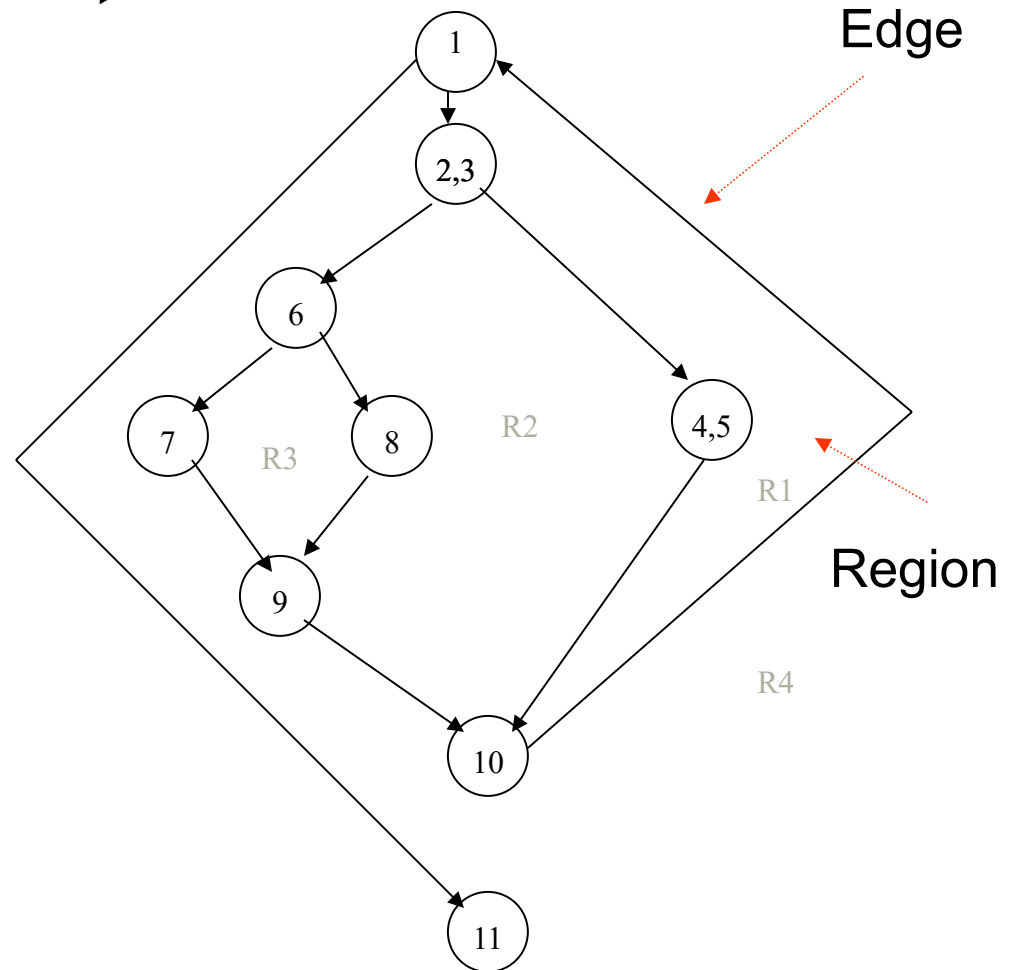
P: number of predicate nodes (simple decisions)
in flow graph

Example-1

Flow chart



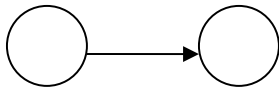
Flow graph



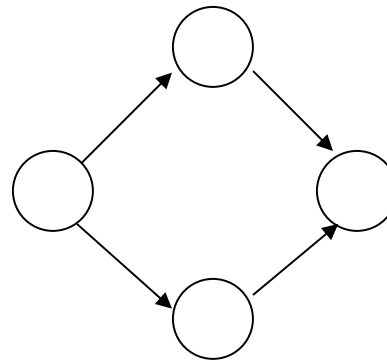
Flow Graph Notation

- Each circle represents one or more nonbranching source code statements.

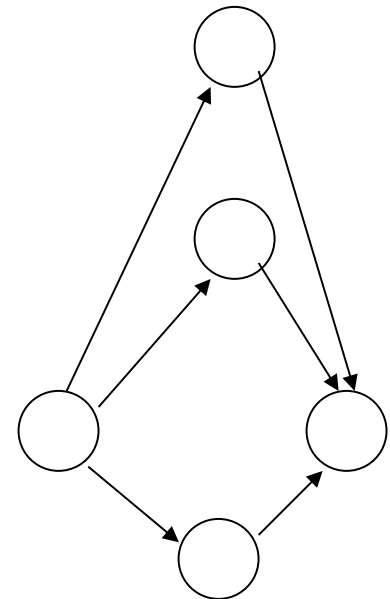
Sequence



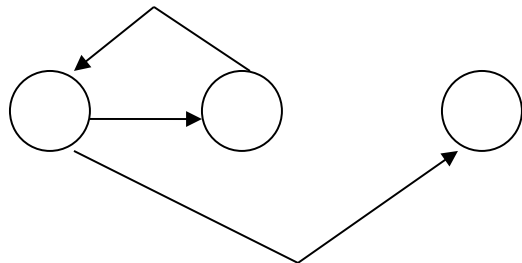
if-else



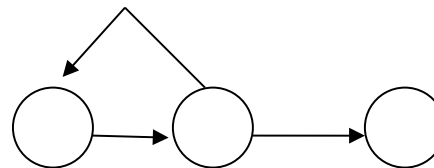
switch



While



Until



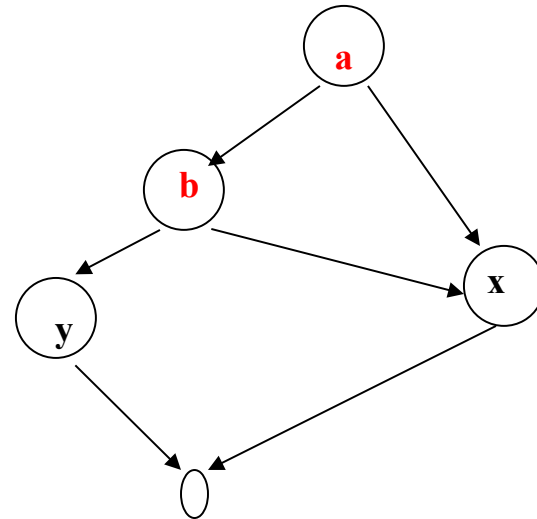
Compound Logic

If **a** OR **b**

then procedure **x**

else procedure **y**

ENDIF



Example-1 (Cyclomatic Complexity)

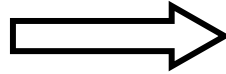
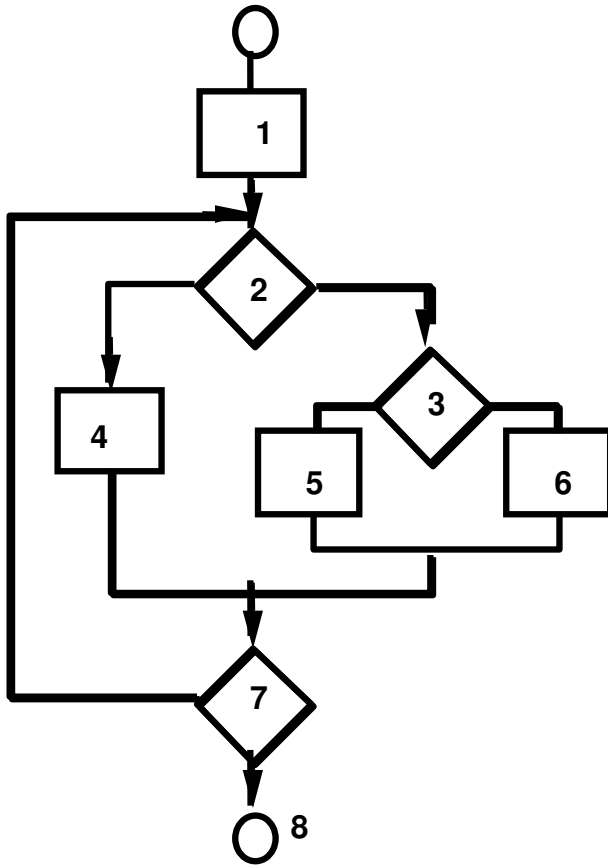
$$\begin{aligned} \infty \quad V(G) &= E - N + 2 \\ &= 6 - 4 + 2 = 4 \end{aligned}$$

∞ **Other method:**

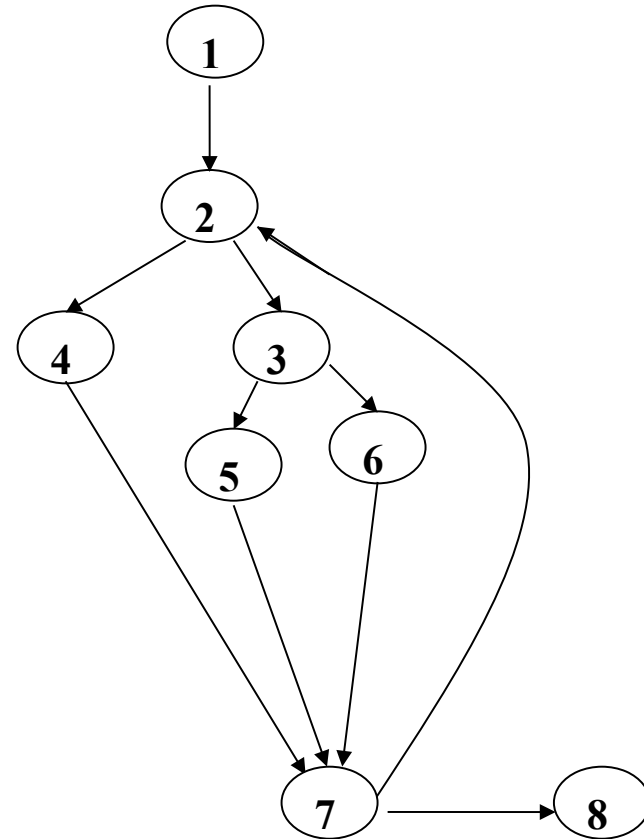
$$\begin{aligned} V(G) &= P + 1 \\ &= 3 + 1 = 4 \end{aligned}$$

Example-2

Flow chart



Flow graph



Example-2 (Cyclomatic Complexity)

First, we calculate $V(G)$:

$$\begin{aligned} V(G) &= E - N + 2 \\ &= 10 - 8 + 2 = 4 \end{aligned}$$

Now, 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

1. Implementation/Programming Guidelines
2. Software Testing Concepts 
3. Unit Testing
4. Module Integration and Testing Strategies
5. Testing Approaches
 1. White-Box Testing
 2. Black-Box Testing
6. Other Types of Testing

Software Testing Concepts

∞11.2 ∞

- Testing is the process of **executing** a program to find errors.
- Testing is planned by defining “Test Cases” in a systematic way.
- **A test case is a collection of input data and expected output.**
- A good test case is one that has a high probability of finding an as-yet-undiscovered error.
- A successful test is one that uncovers an as-yet-undiscovered error.

Principles

- ✧ Most of the tests should be traceable to requirements.
 - Both customer and system requirements
- ✧ Tests should be planned and implemented long before testing begins.
 - After requirements model is complete.
 - Except unit testing, because units should be implemented before the tests in a test-last approach.
- ✧ Exhaustive testing is impossible.
- ✧ Testing should be conducted by the developer of the software and also by an independent test group.

Verification and Validation

- ∞ **Verification**: Are we building the product right?
- ∞ **Validation**: Are we building the right product?

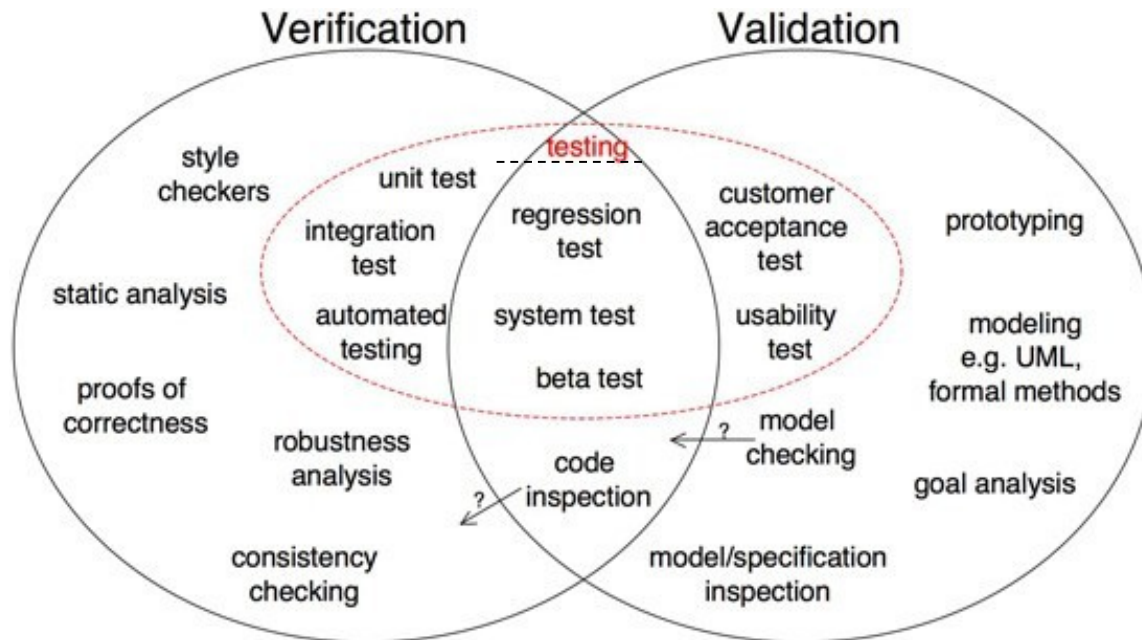


Figure from S. Easterbrook 2010

Phases of Testing

1. Unit Testing

Does each module do what it supposed to do?

2. Integration Testing

Do you get the expected results when the parts are put together?

3. System Testing

Does it work within the overall system?

Does the program satisfy the requirements ?

Levels of Testing

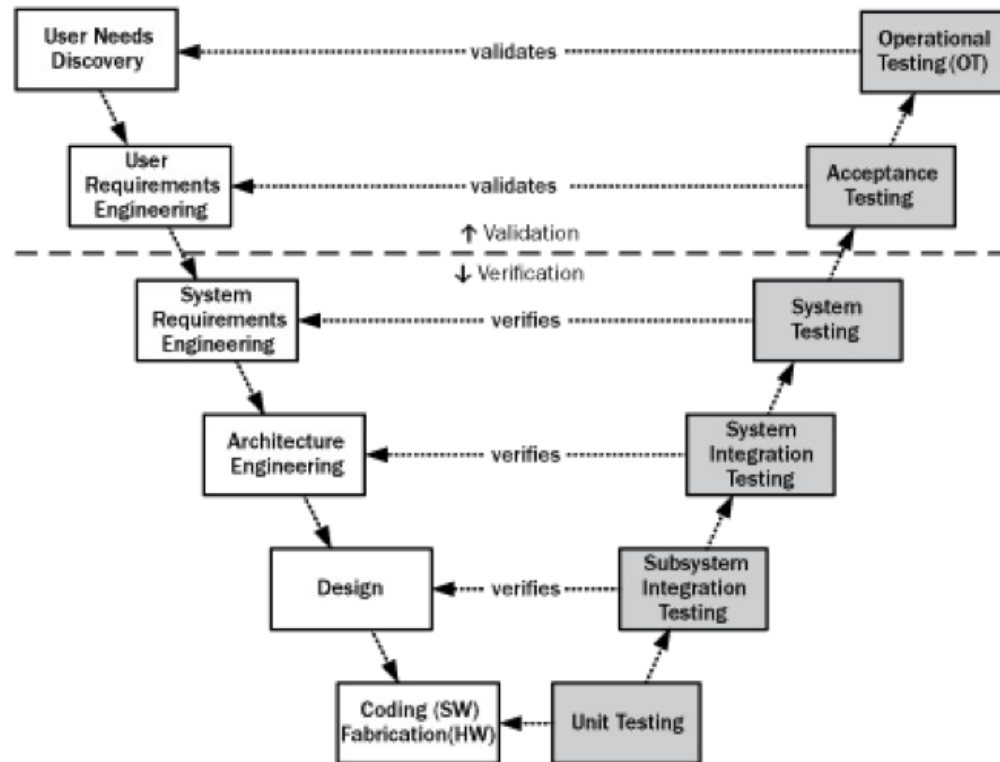


Figure from sei.cmu.edu

Levels of testing in Agile

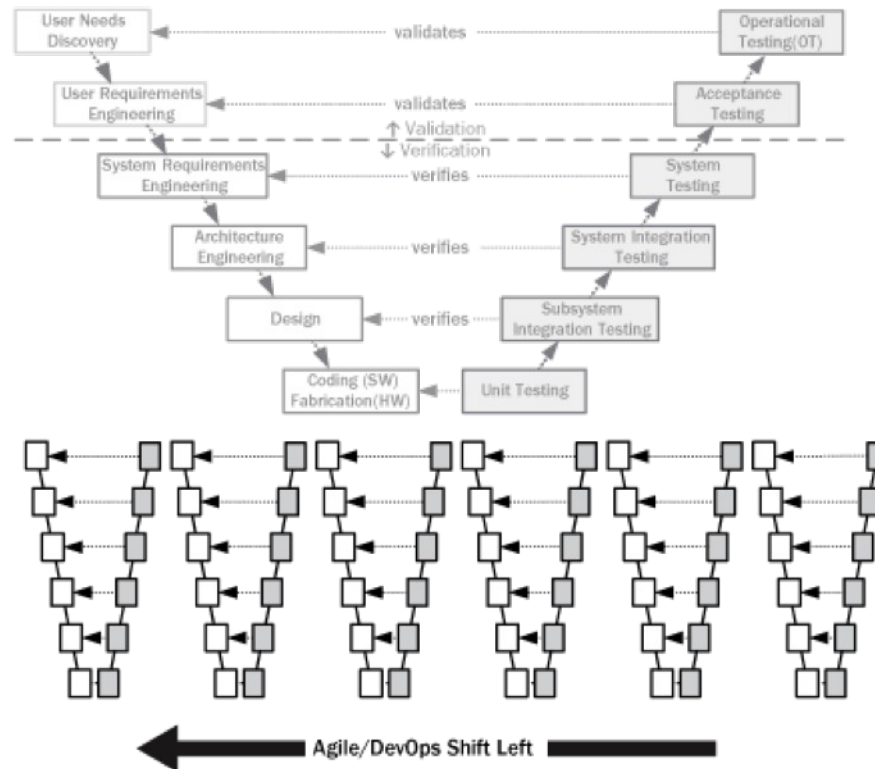



Figure from sei.cmu.edu

1. Implementation/Programming Guidelines
2. Software Testing Concepts
3. Unit Testing 
4. Module Integration Testing
5. Testing Approaches
 1. White-Box Testing
 2. Black-Box Testing
6. Other Types of Testing

Unit Testing

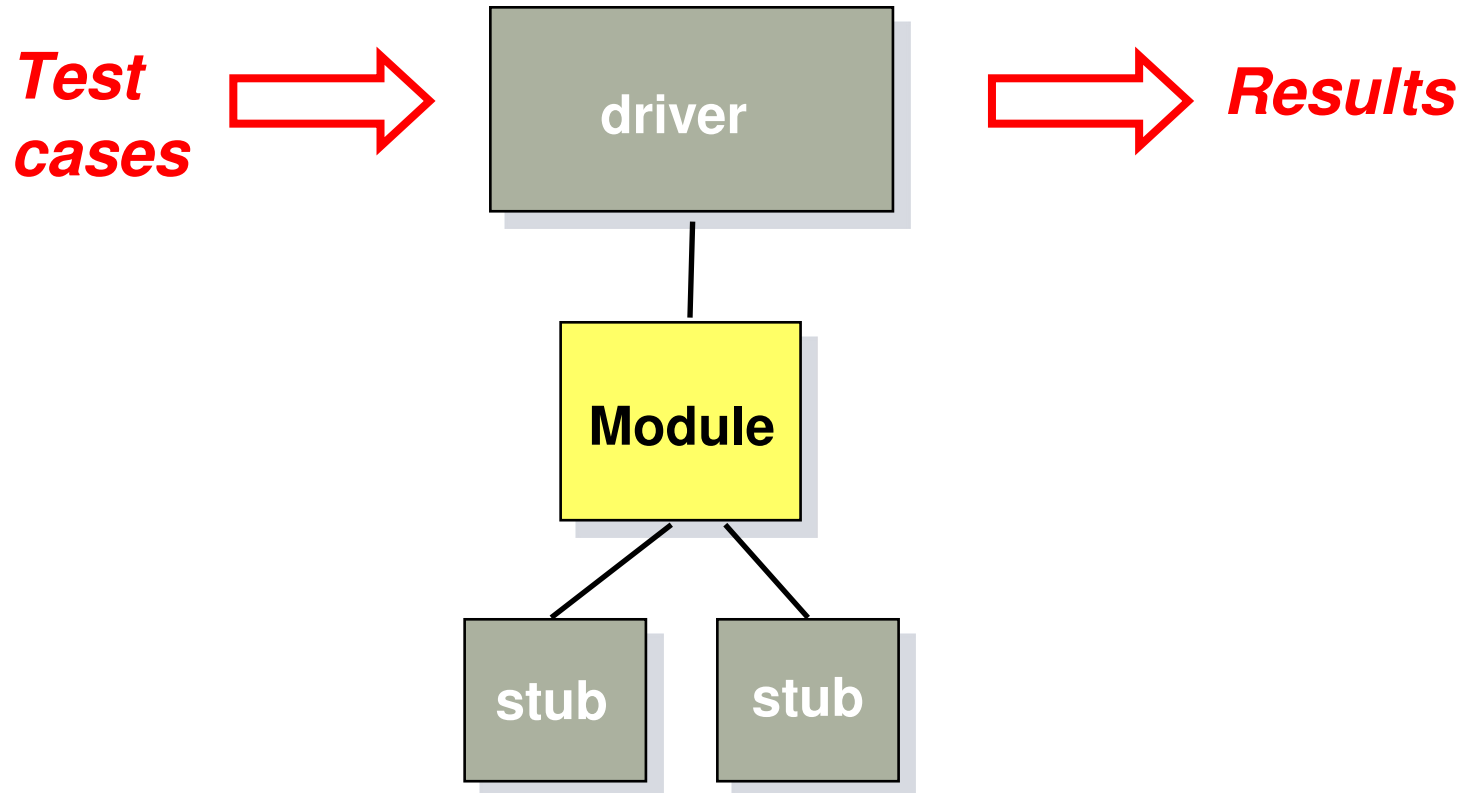
∞11.3∞

- ∞ Testing the smallest units of the code (methods, classes)
- ∞ Dynamic execution:
 - Test the **input/output** behavior
 - Also test the internal **logic**
- ∞ There are other static analysis techniques that are not considered as testing.
 - Hand execution/Code review (Reading the source code)
 - Code inspection (also known as Walkthrough)
 - Automated tools checking for syntactic and semantic errors (FindBugs)

Unit testing

- ✎ Unit testing is done by programmers, for programmers.
- ✎ Unit tests are performed to prove that a piece of code does what the developer thinks it should do.
- ✎ When there are dependencies between the units, it is the programmers' responsibility to reduce such dependencies as much as possible, or to use additional strategies for testing the module as a whole.

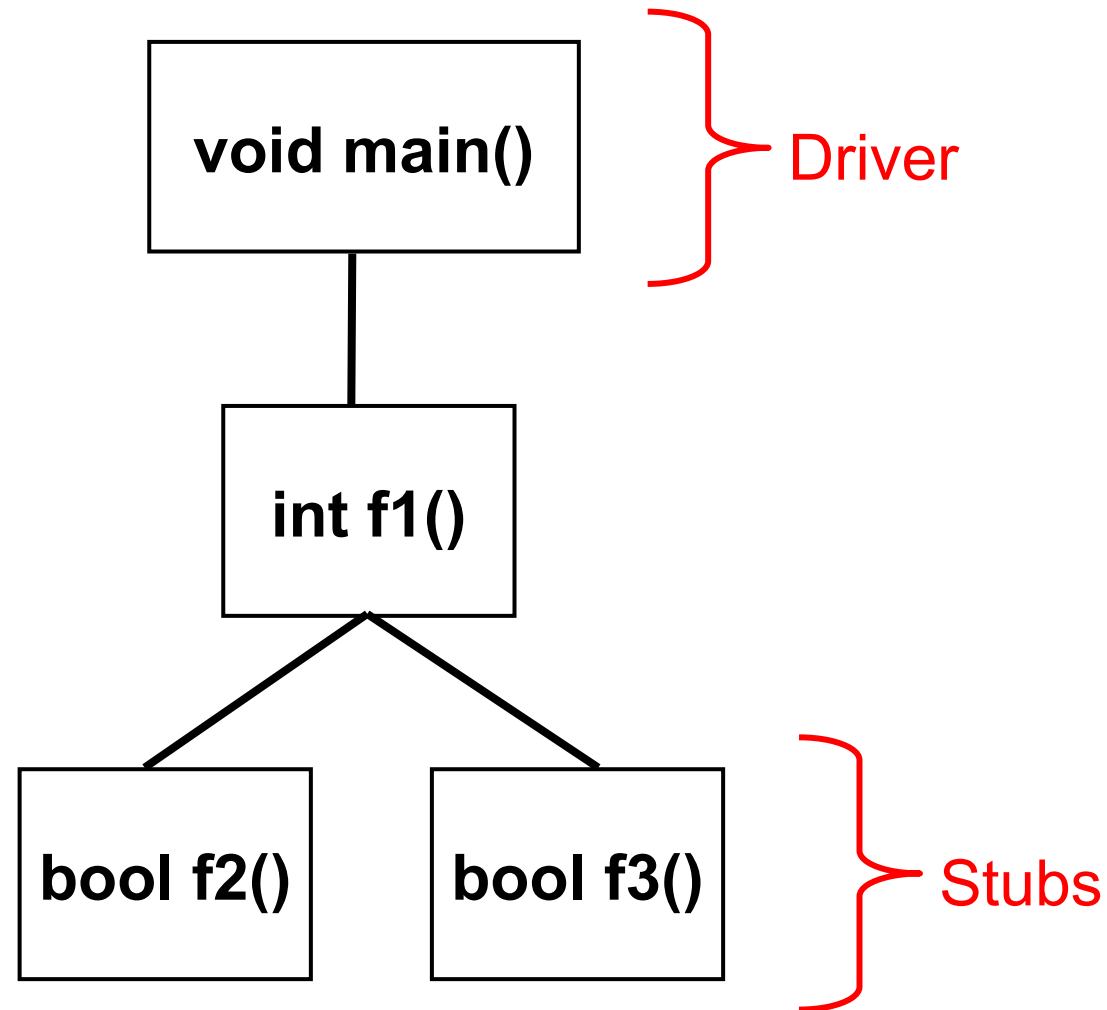
- Stubs and drivers should be written in integration/unit testing.



Stub modules and driver modules

- ✎ A stub is an empty (dummy) module, which:
 - Just prints a message ("Module X called"), or
 - Returns pre-determined values from pre-planned test cases.
 - Used in top-down strategy
- ✎ A driver is a caller module, which calls its stubs:
 - Once or several times,
 - Each time checking the value returned.
 - Used in bottom-up strategy

Example: Stubs and driver (1)



Example: Stubs and driver (2)


```
#include <stdio.h>
#include <math.h>
#define TRUE 1
#define FALSE 0

// f2 is a stub
bool f2(int X)
{
    return TRUE;
}

// f3 is a stub
bool f3(int X)
{
    return TRUE;
}
```

```
// We are testing f1
int f1(int X)
{
    if (f2(X) || f3(X))
        printf("%f", sqrt(X));
    else
        printf("Invalid value for X");
}

// This is the driver.
void main()
{
    f1(49);
}
```

1. Implementation/Programming Guidelines
2. Software Testing Concepts
3. Unit Testing
4. Module Integration Testing 
5. Testing Approaches
 1. White-Box Testing
 2. Black-Box Testing
6. Other Types of Testing

Module Integration Testing and Strategies

11.4

A strategic approach to Testing

- ∞ Testing begins at the module level and works outward toward the integration of the entire system.
- ∞ Different testing techniques are appropriate at different points in time.

Strategies for Integration Testing

1) Big bang approach:

- All modules are fully implemented and combined as a whole, then tested as a whole. It is not practical.

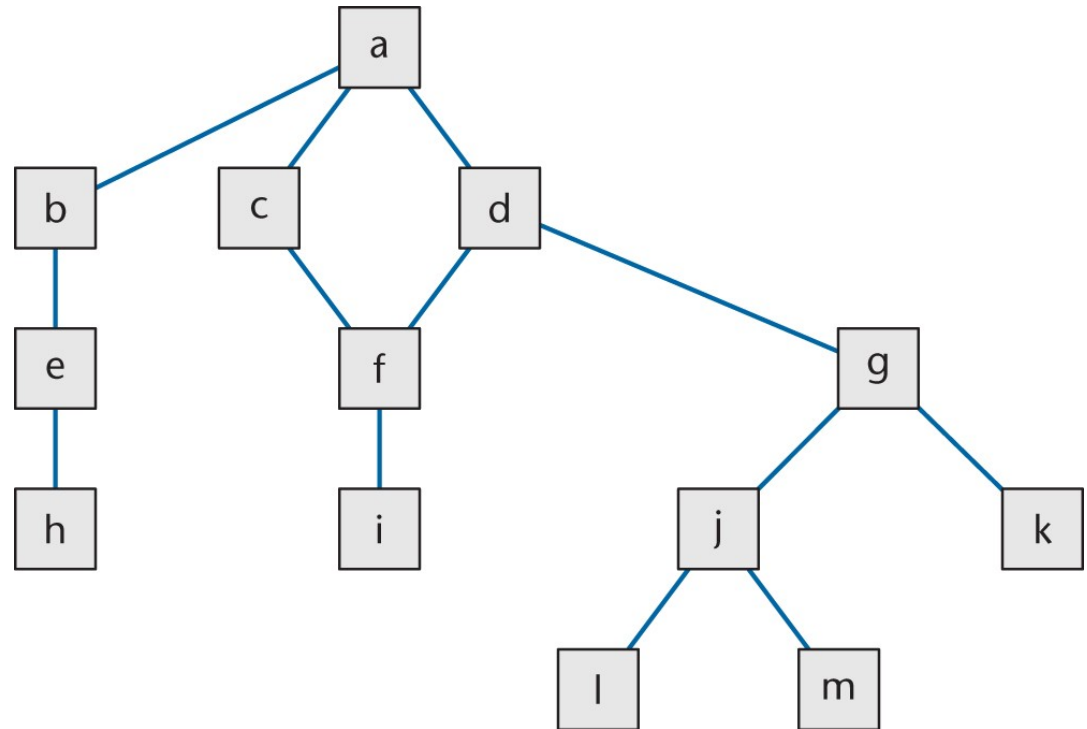
2) Incremental approach: (Top-down or Bottom-up)

- Program is constructed and tested in small clusters.
- Errors are easier to isolate and correct.
- Interfaces between modules are more likely to be tested completely.
- After each integration step, a regression test is conducted.

Big bang integration testing

Integration and testing
at once:

1. a, b, c, ..., l, m

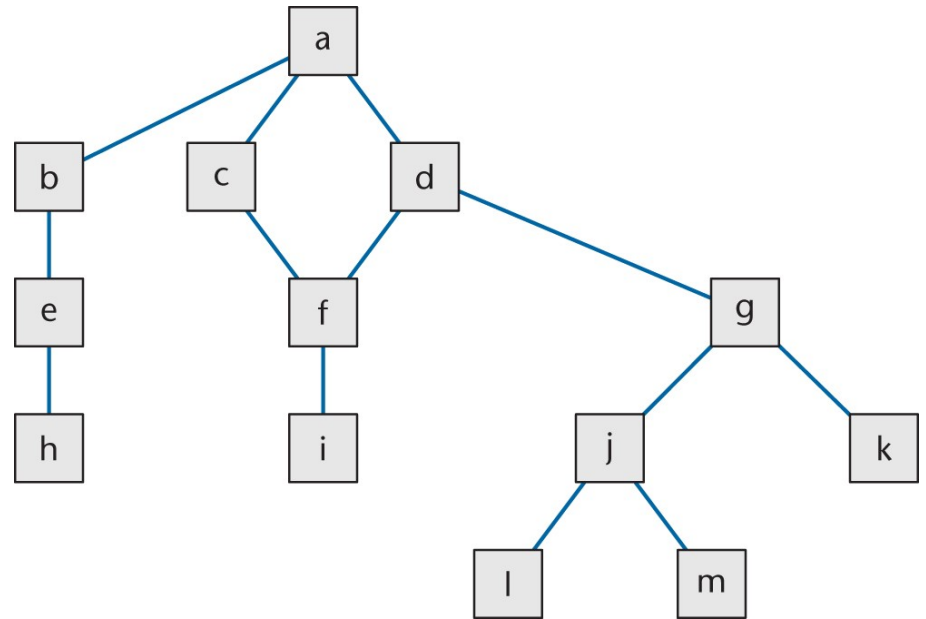


Top-down Integration

∞ If code artifact `mAbove` sends a message to artifact `mBelow`, then `mAbove` is implemented and integrated before `mBelow`

∞ One possible top-down ordering is

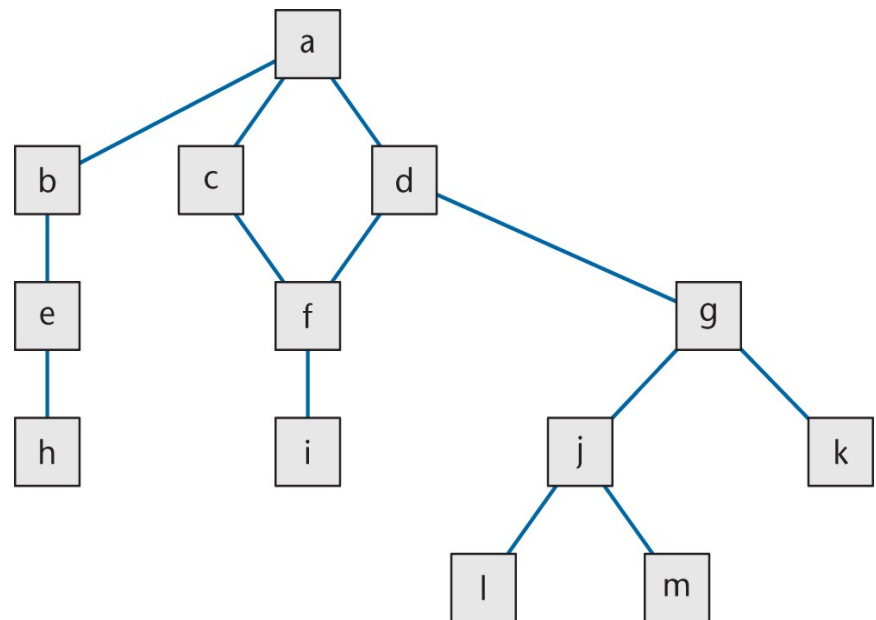
- a, b, c, d, e, f, g, h, i, j, k, l, m



Top-down Integration (contd)

Another possible top-down ordering is

	a
[a]	b, e, h
[a]	c, d, f, i
[a, d]	g, j, k, l, m

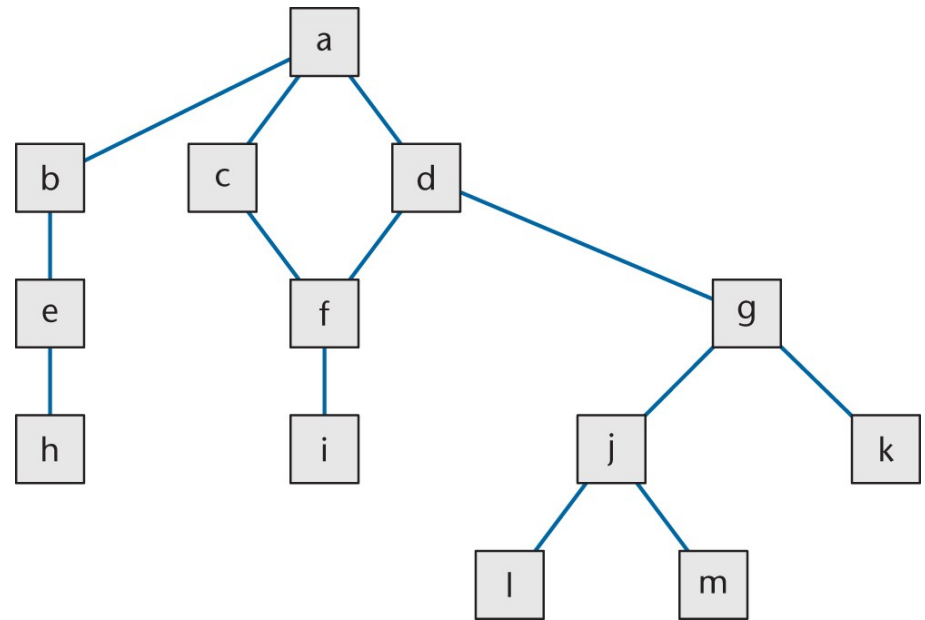


Bottom-up Integration

∞ If code artifact m_{Above} calls code artifact m_{Below} , then m_{Below} is implemented and integrated before m_{Above}

∞ One possible bottom-up ordering is

l, m,
h, i, j, k,
e, f, g,
b, c, d,
a



Bottom-up Integration

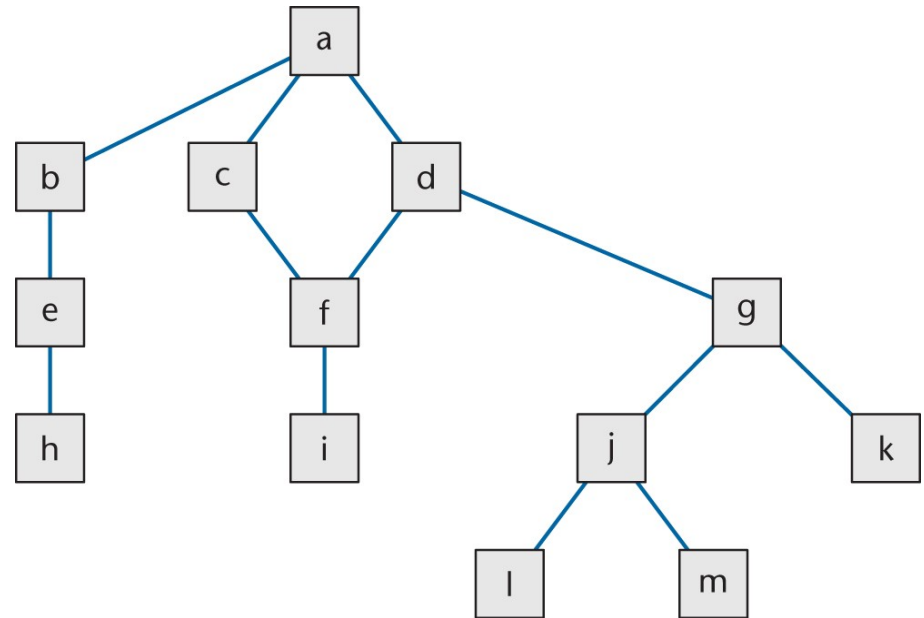
Another possible
bottom-up ordering is

h, e, b

i, f, c, d

l, m, j, k, g [d]

a [b, c, d]



Sandwich Integration

- Logic artifacts are integrated top-down
- Operational artifacts are integrated bottom-up
- Finally, the interfaces between the two groups are tested

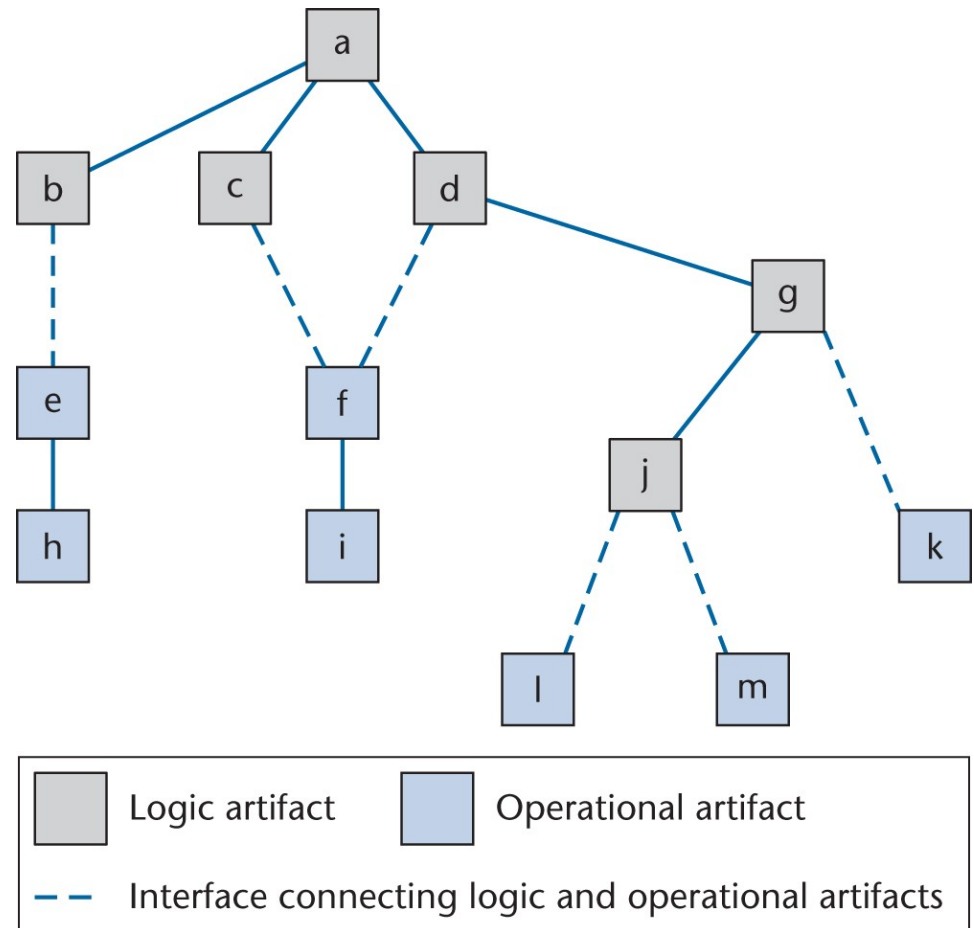



Figure 15.7

1. Implementation/Programming Guidelines
2. Software Testing Concepts
3. Unit Testing
4. Module Integration and Testing Strategies
5. Testing Approaches 
 1. White-Box Testing
 2. Black-Box Testing
6. Other Types of Testing

Testing Approaches

∞11.5∞

∞ Black-box testing

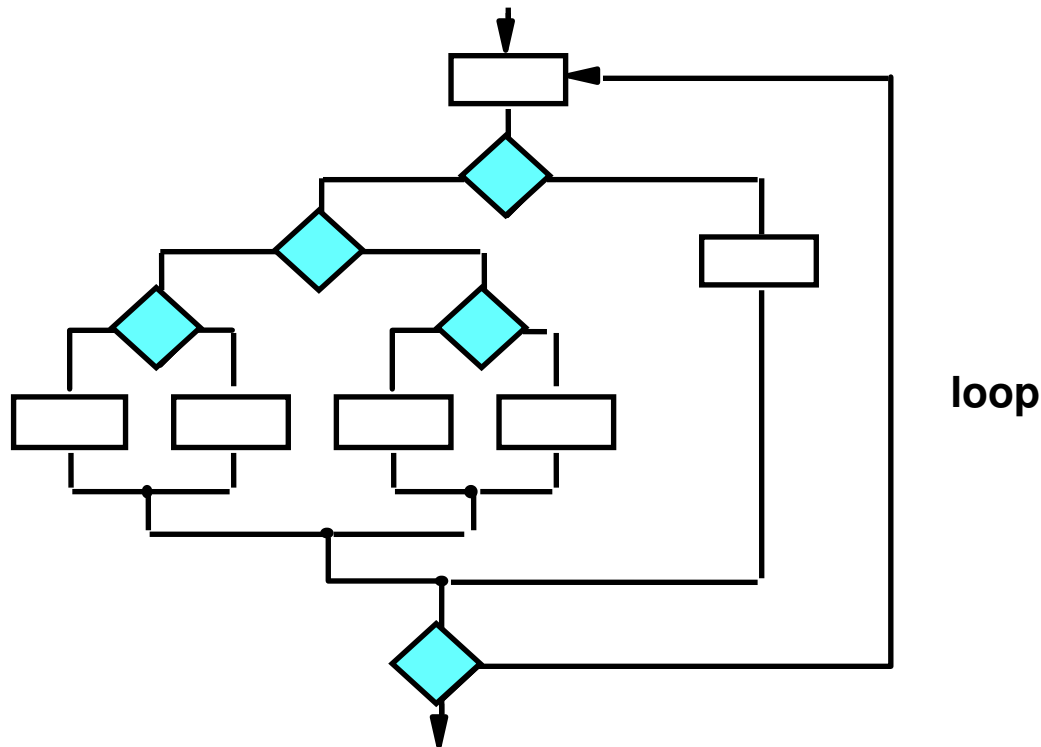
- Testing of the software interfaces
- Used to demonstrate that
 - **functions** are operational
 - **input** is properly accepted and **output** is correctly produced

∞ White-box testing

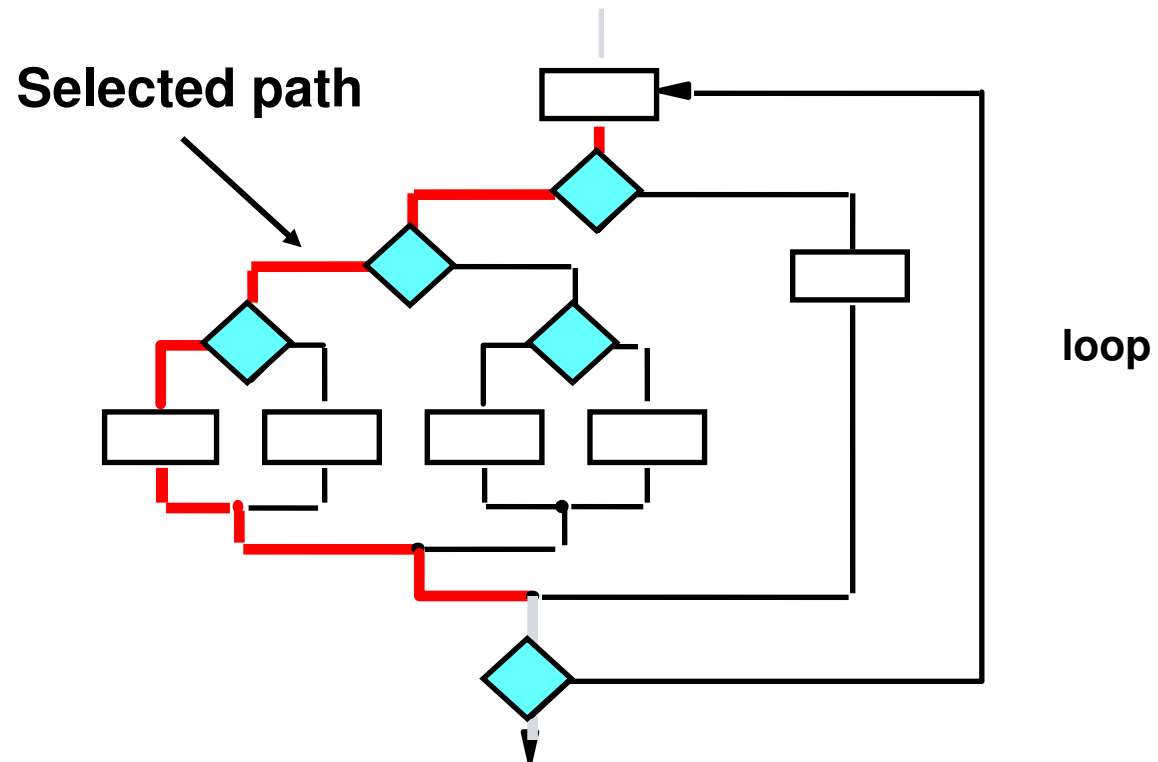
- Close examination of procedural details
- Logical paths through the software is tested by test cases that exercise specific sets of conditions and loops

Exhaustive Testing

- There are 5 separate paths inside the following loop.
- Assuming loop ≤ 20 , there are $5^{20} \approx 10^{14}$ possible paths.
- Exhaustive testing is not feasible.



∞ Instead of exhaustive testing, a selective testing is more feasible.



Testing to Specifications versus Testing to Code

- ✎ There are two extremes to testing
- ✎ *Test to code* (also called glass-box, logic-driven, structured, or path-oriented testing)
 - Ignore the specifications — use the code to select test cases
- ✎ *Test to specifications* (also called black-box, data-driven, functional, or input/output driven testing)
 - Ignore the code — use the specifications to select test cases

Example: Script for Unit Testing

- 🌀 This is the format for a test plan to show what you're planning to do.
- 🌀 It should to be filled to show what happened when you run tests.

Scenario # : 1 Tester: AAA BBB Date of Test: 01/01/2024				
Test Number	Test Description / Input	Expected Result	Actual Result	Fix Action
1	Invalid file name	"Error: File does not exist"		
2	Valid filename, but file is binary	"Error: File is not a text file"		
3	Valid filename	"Average = 99.00"		
4				
5				

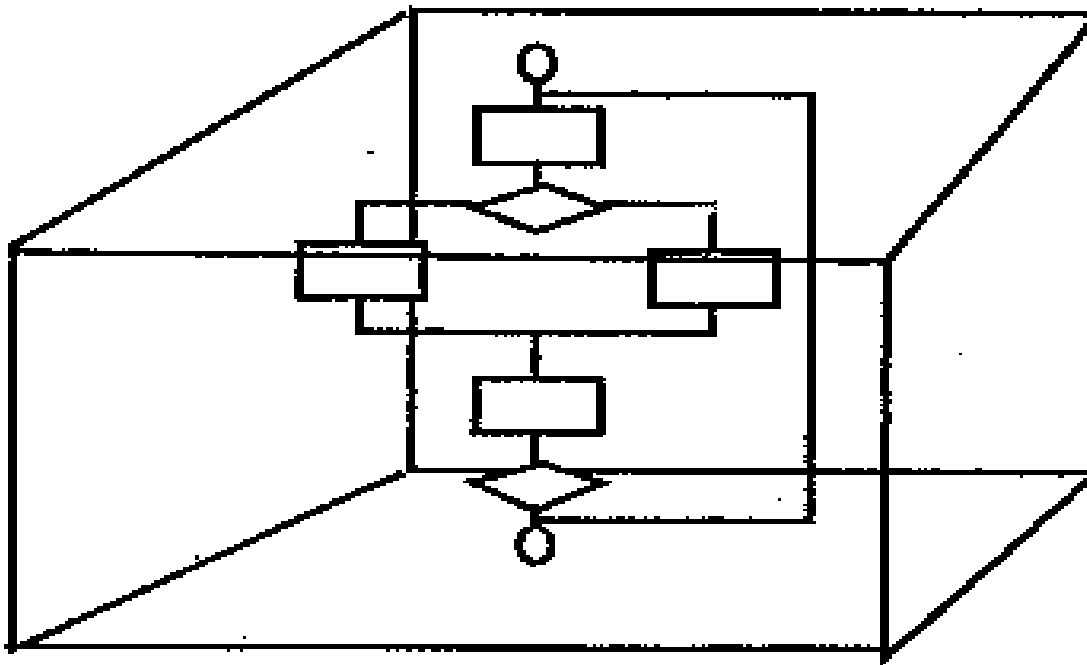
1. Implementation/Programming Guidelines
2. Software Testing Concepts
3. Unit Testing
4. Module Integration and Testing Strategies
5. Testing Approaches
 1. White-Box Testing ←
 2. Black-Box Testing
6. Other Types of Testing

White-Box Testing

11.5.1

White-Box Testing

- ⌘ Our goal is to ensure that all statements and conditions have been executed at least once.
- ⌘ **Code coverage:** At a minimum, every line of code should be executed by at least one test case.

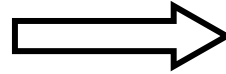
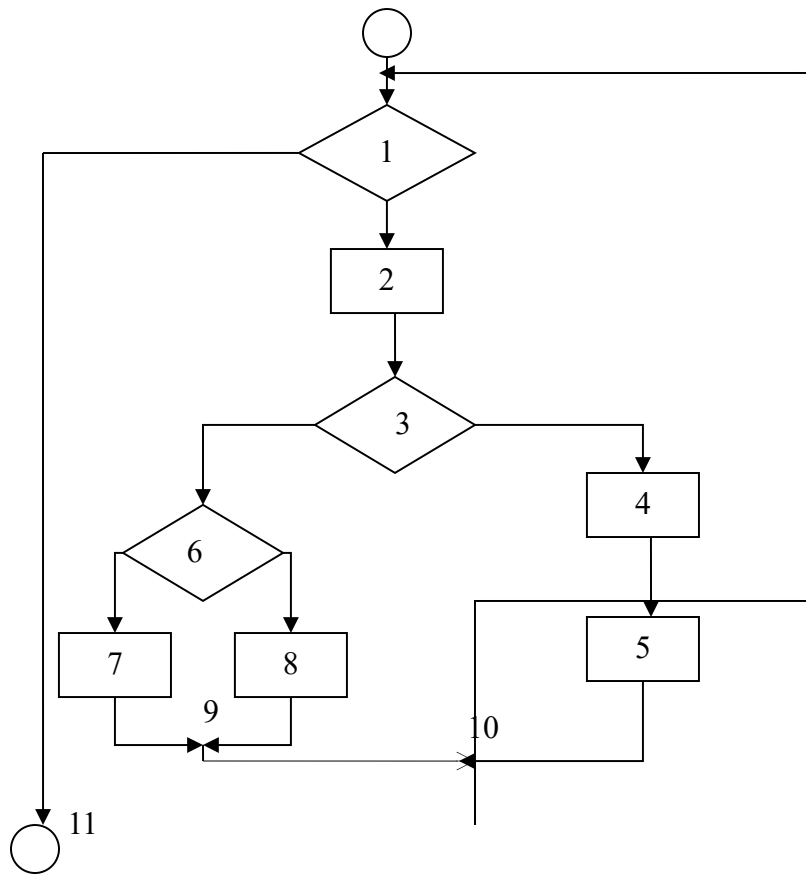


Flow graph for White box testing

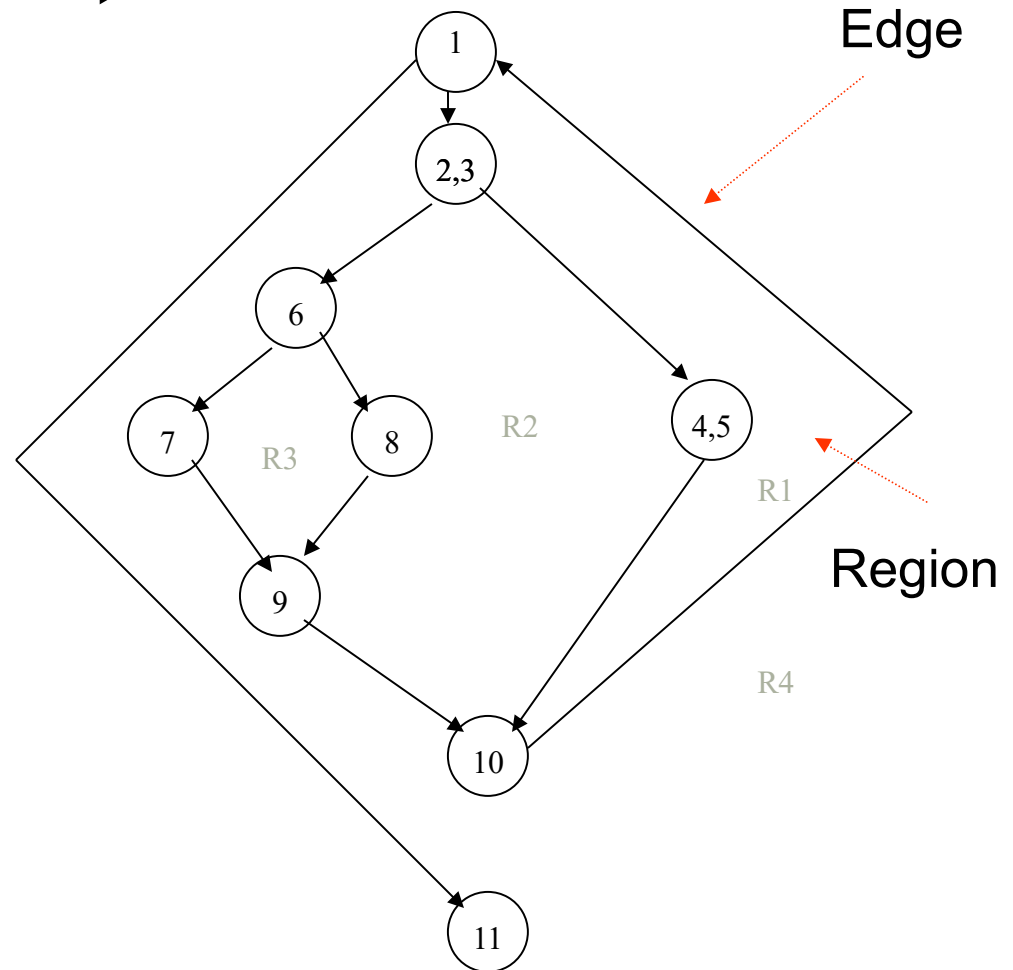
- ∞ To help the programmer systematically test the code
 - Each branch in the code (such as if and while statements) creates a node in the graph
 - The testing strategy has to reach a targeted coverage of source code; the objective can be to:
 - cover all possible paths (often infeasible)
 - cover all possible nodes/statements (simpler)
 - cover all possible edges/branches (most efficient)

Example-1

Flow chart



Flow graph



Coverage in White-box testing

- Although 100% coverage is an admirable goal, 100% of the wrong type of coverage can lead to problems.

T,T,T would satisfy
%100 statement
coverage but does
not cover all
branches.
F,F,F and T,T,T

```
public class Path {  
    public int returnInput(int x,  
                           int boolean,  
                           int boolean,  
                           int boolean)  
    {  
        if(condition1){  
            x++;  
        }  
        if(condition2){  
            x--;  
        }  
        if(condition3){  
            x=x;  
        }  
        return x;  
    }  
}
```

Coverage in White-box testing

- ✎ Write test inputs for %100 statement coverage
- ✎ Now for %100 branch coverage
- ✎ Are these inputs able to catch the defects?

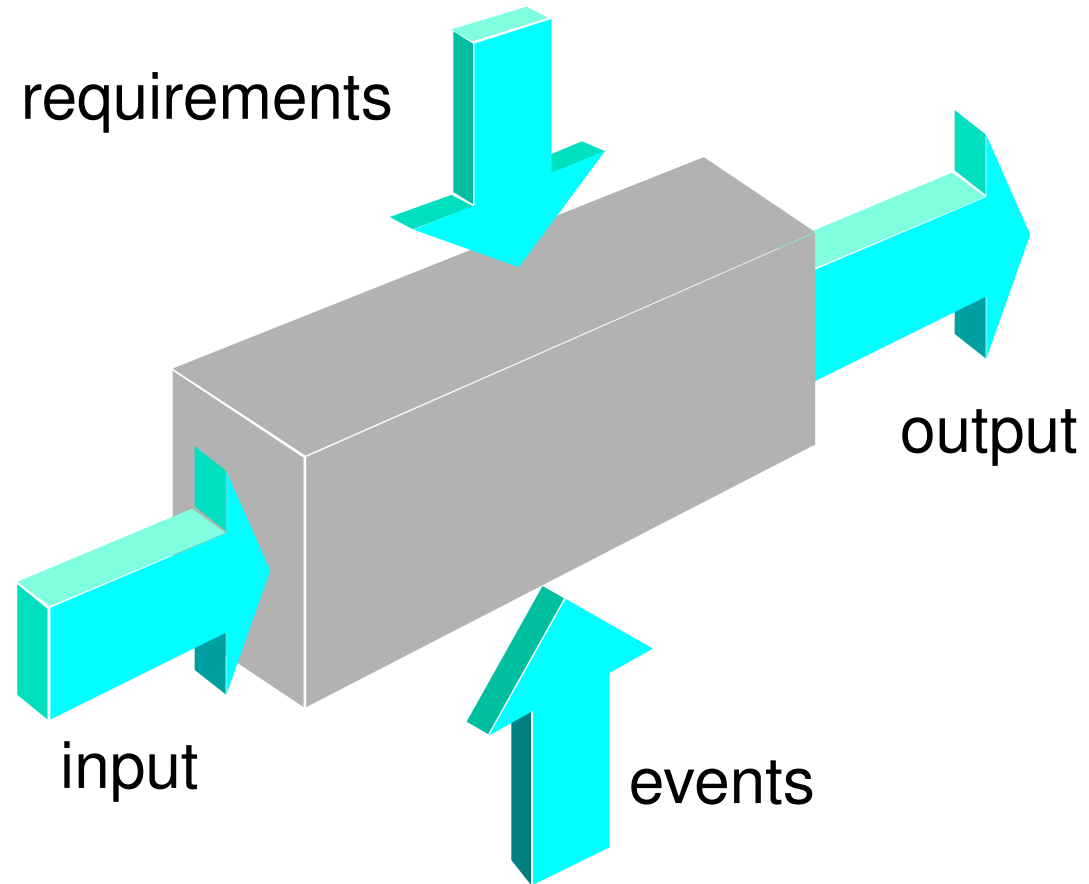
```
void main() {  
    float x,y;  
    read(x);  
    read(y);  
    if( x== 0 || y >0)  
        y = y/x;  
    else  
        x = y + 2;  
    write(x);  
    write(y);  
}
```

1. Implementation/Programming Guidelines
2. Software Testing Concepts
3. Unit Testing
4. Module Integration and Testing Strategies
5. Testing Approaches
 1. White-Box Testing
 2. Black-Box Testing
6. Other Types of Testing



Black-Box Testing

11.5.20



Black Box Testing

- ✧ Black-box testing is conducted at the software interface to demonstrate that correct inputs results in correct outputs.
- ✧ Testing software against a specification of its external behavior without knowledge of internal implementation details
- ✧ It is not an alternative to White Box testing, but complements it.
- ✧ Focuses on the functional requirements.
- ✧ Attempts to find errors on
 - Incorrect or missing functions
 - Interface errors
 - Errors in data structures or external database access
 - Performance errors
 - Initialization and termination errors

Valid data:

- user supplied commands
- responses to system prompts
- file names
- computational data
 - physical parameters
 - bounding values
 - initiation values
- responses to error messages
- mouse picks

Invalid data:

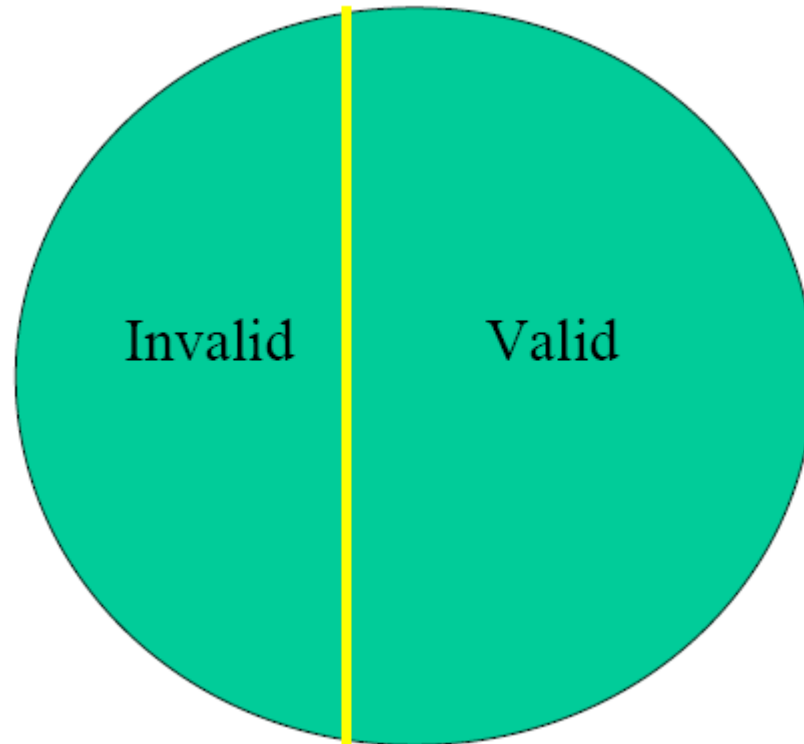
- data outside bounds of the program
- physically impossible data
- proper value supplied in wrong place

Examples of Input Domain

- ∞ Boolean value
 - True or False
- ∞ Numeric value in a particular range
 - $99 \leq N \leq 999$
 - Integer, Floating point
- ∞ One of a fixed set of enumerated values
 - {Jan, Feb, Mar, ...}
 - {VisaCard, MasterCard, DiscoverCard, ...}
- ∞ Formatted strings
 - Phone numbers
 - File names
 - URLs
 - Credit card numbers

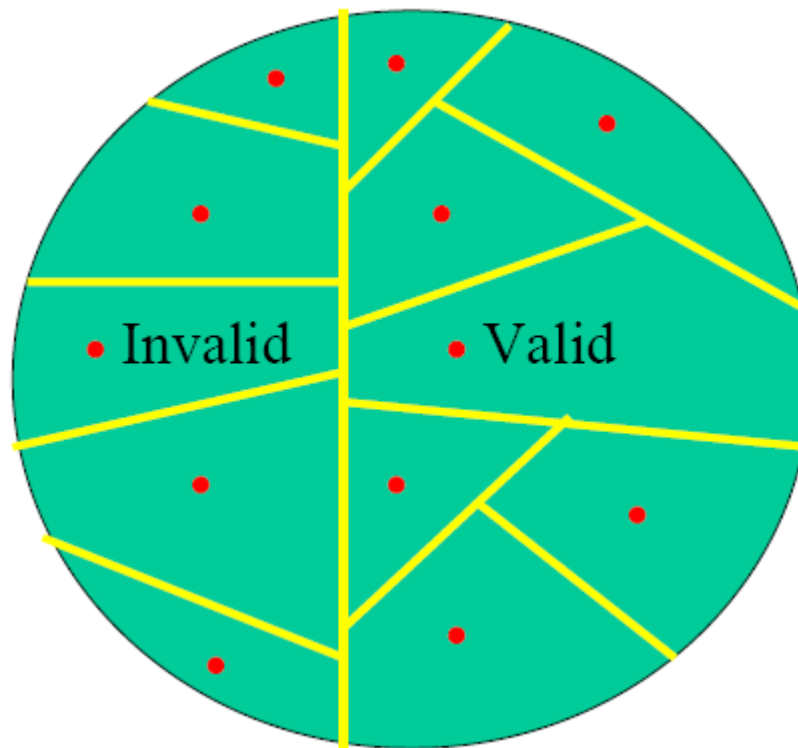
Equivalence Partitioning (1)

✎ First-level partitioning: Valid vs. Invalid input values



Equivalence Partitioning (2)


- ✎ Partition valid and invalid values into equivalence classes
- ✎ Create a test case for at least one value from each equivalence class



Equivalence Partitioning - Examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: -99 <= N <= 99	?	?
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?	?

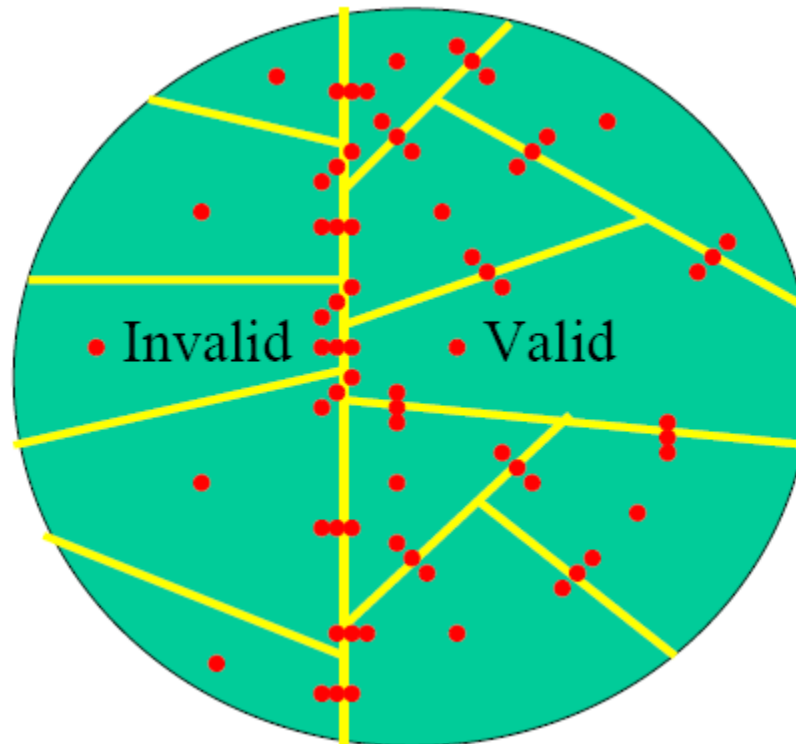
Equivalence Partitioning - Examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: -99 <= N <= 99	[-99, -10] [-9, -1] 0 [1, 9] [10, 99]	< -99 > 99 Malformed numbers {12-, 1-2-3, ...} Non-numeric strings {junk, 1E2, \$13} Empty value
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	 (555)555-5555 555-555-5555 200 <= Area code <= 999 200 < Prefix <= 999	Area code < 200 Area code > 999 Area code with non-numeric characters <i>Similar for Prefix and Suffix</i> Invalid format 5555555, (555)(555)5555, etc.

- ☞ When choosing values to test, use the values that are most likely to cause the program to fail
- ☞ If (200 < areaCode && areaCode < 999)
 - Testing area codes 200 and 999 would catch the coding error
- ☞ In addition to testing center values, we should also test boundary values
 - Right on a boundary
 - Very close to a boundary on either side

Boundary Value Analysis (3)

- ✎ Create test cases to test boundaries between equivalence classes.



Boundary Value Analysis - Examples

Input	Boundary Cases
A number N such that: $-99 \leq N \leq 99$?
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?

Boundary Value Analysis - Examples

Input	Boundary Cases
A number N such that: $-99 \leq N \leq 99$	-100, -99, -98 -10, -9 -1, 0, 1 9, 10 98, 99, 100
Phone Number Area code: [200, 999] Prefix: (200, 999) Suffix: Any 4 digits	Area code: 199, 200, 201 Area code: 998, 999, 1000 Prefix: 200, 199, 198 Prefix: 998, 999, 1000 Suffix: 3 digits, 5 digits

Example:

Function Calculating Average

Function Calculating Average

```
// The minimum value is excluded from average.  
float Average (float scores [ ] , int length)  
{  
    float min = 99999;  
    float total = 0;  
    for (int i = 0; i < length; i++)  
    {  
        if (scores[i] < min)  
            min = scores[i];  
  
        total += scores[i];  
    }  
    total = total - min;  
    return total / (length - 1);  
}
```

Test Cases (Basis: **Array Length**)

Test case (input)	Basis: Array length				Expected output	Notes
	Empty	One	Small	Large		
()	x				0.0	crashes!
(87.3)		x			87.3	
(90,95,85)			x		92.5	
(80,81,82,83, 84,85,86,87, 88,89,90,91)				x	86.0	

Test Cases (Basis: **Position of Minimum**)

Test case (input)	Basis: Position of minimum			Expected output	Notes
	First	Middle	Last		
(80,87,88,89)	x			88.0	
(87,88,80,89)		x		88.0	
(99,98,0,97,96)		x		97.5	
(87,88,89,80)			x	88.0	

Test Cases (Basis: **Number of Minima**)

Test case (input)	Basis: Number of minima			Expected output	Notes
	One	Several	All		
(80,87,88,89)	x			88.0	
(87,86,86,88)		x		87.0	
(99,98,0,97,0)		x		73.5	
(88,88,88,88)			x	88.0	

Example:

Function Normalizing an Array

Function Normalizing an Array

- ✎ The following C function is intended to normalize an array.
- ✎ Normalization Algorithm:
 - First, the array of N elements is searched for the smallest and largest non-negative elements.
 - Secondly, the range is calculated as $\text{max} - \text{min}$.
 - Finally, all non-negative elements that are bigger than the range are normalized to 1, or else to 0.
- ✎ There are no syntax errors, but there may be logic errors in the following implementation.

C function

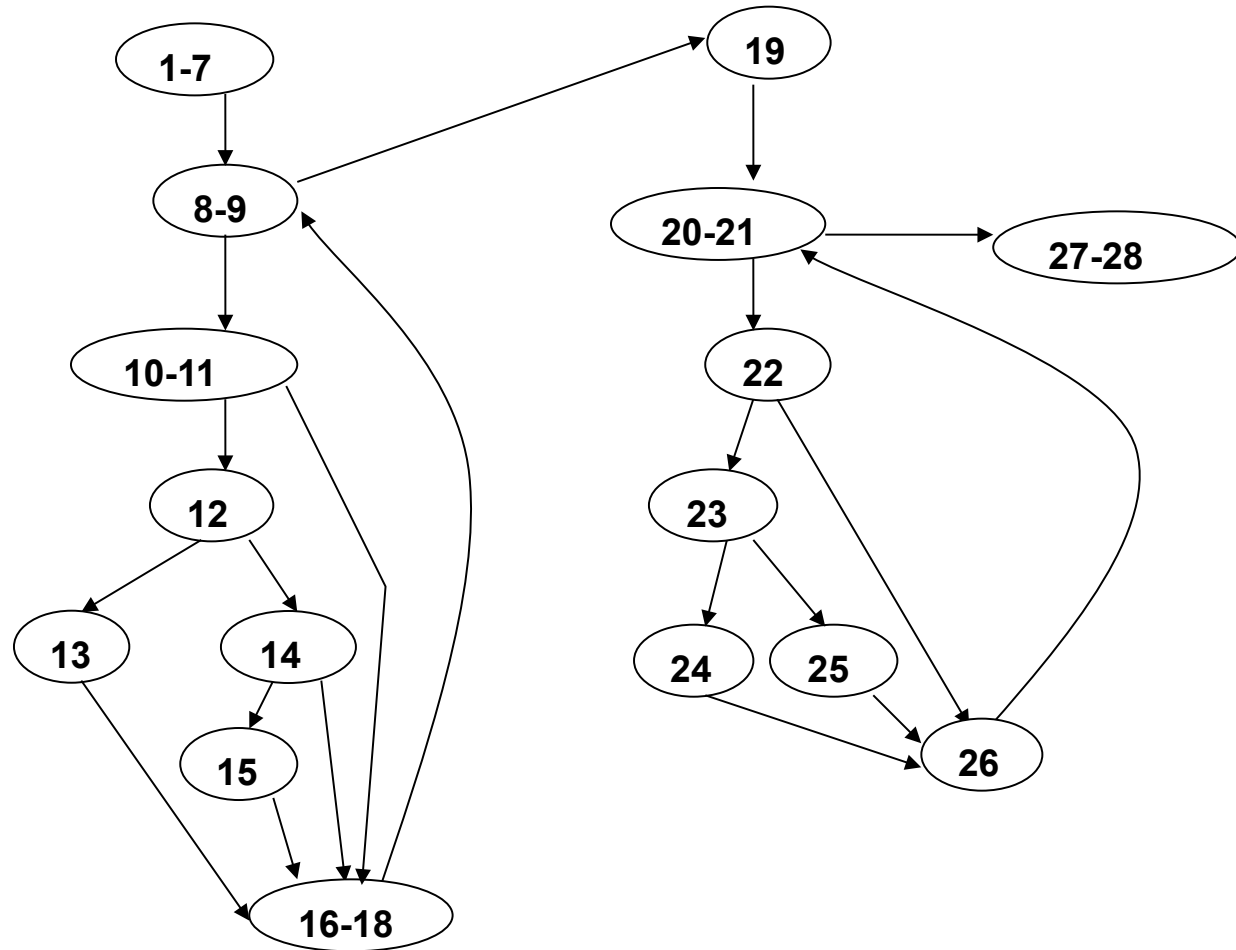
```
1. int normalize (int A[], int N)
2. {
3.     int range, max, min, i, valid;
4.     range = 0;
5.     max = -1;
6.     min = -1;
7.     valid = 0;
8.     for (i = 0; i < N; i++)
9.     {
10.         if (A[i] >= 0)
11.         {
12.             if (A[i] > max)
13.                 max = A[i];
14.             else if (A[i] < min)
15.                 min = A[i];
16.             valid++;
17.         }
18. }
```

```
19. range = max - min;
20. for (i = 0; i < N; i++)
21. {
22.     if (A[i] >= 0)
23.         if (A[i] >= range)
24.             A[i] = 1;
25.         else A[i] = 0;
26. }
27. return valid;
28. }
```

Tasks

- ✎ Draw the corresponding flow graph and calculate the cyclomatic complexity $V(g)$.
- ✎ Find linearly independent paths for basis path testing.
- ✎ Give test cases for boundary value analysis.

Flow Graph



Cyclomatic Complexity and Test Paths

Number of edges = $E = 21$

Number of nodes = $N = 16$

Cyclomatic complexity = $V(g) = E - N + 2 = 21 - 16 + 2 = 7$

Path1: 1-7, 8-9, 10-11, 12, 13, 16-18, 19, 20-21, 22, 23, 24, 26, 27-28

Path2: 1-7, 8-9, 10-11, 12, 14, 15, 16-18, 19, 20-21, 22, 23, 24, 26, 27-28

Path3: 1-7, 8-9, 10-11, 12, 14, 16-18, 19, 20-21, 22, 23, 24, 26, 27-28

Path4: 1-7, 8-9, 19, 20-21, 22, 23, 24, 26, 27-28

Path5: 1-7, 8-9, 10-11, 12, 13, 16-18, 19, 20-21, 22, 23, 25, 26, 27-28

Path6: 1-7, 8-9, 10-11, 12, 13, 16-18, 19, 20-21, 22, 26, 27-28

Path7: 1-7, 8-9, 10-11, 12, 13, 16-18, 19, 20-21, 27-28

Test cases for boundary value analysis

Test Case-1)

Input Condition: $N=0$, $A[] = \{10, 20, 30, 40, 50\}$

Expected Output: Valid =0 (Due to invalid N value)

Test Case-2)

Input Condition: $N=4$, $A[] = \{10, 20, 30, 40, 50\}$

Expected Output: Valid =4 $A[] = \{0, 0, 1, 1, 50\}$

Test Case-3)

Input Condition: $N=5$, $A[] = \{10, 20, 30, 40, 50\}$

Expected Output: Valid =5, $A[] = \{0, 0, 0, 1, 1\}$

Test cases for boundary value analysis

Test Case-4)

Input Condition: $N=5$, $A[] = \{-10, -20, -30, -40, -50\}$

Expected Output: Valid =0

Test Case-5)


Input Condition: $N=5$, $A[] = \{0, 0, 0, 0, 0\}$

Expected Output: Valid =5, $A[] = \{1, 1, 1, 1, 1\}$

Test Case-6)

Input Condition: $N=5$, $A[] = \{10, 10, 10, 10, 10\}$

Expected Output: Valid =5, $A[] = \{1, 1, 1, 1, 1\}$

1. Implementation/Programming Guidelines
2. Software Testing Concepts
3. Unit Testing
4. Module Integration and Testing Strategies
5. Testing Approaches
 1. White-Box Testing
 2. Black-Box Testing
6. Other Types of Testing 

Other Types of Testing

❧11.6❧

- Regression testing
- Alpha test and beta test
- Performance testing
- Volume testing
- Stress testing
- Security testing
- Usability testing
- Recovery testing
- Testing web-based systems

1. Functional Testing
2. Performance Testing
3. Installation Testing

Functional Testing

- ✎ An alternative form of black-box testing for classical software
 - We base the test data on the functionality of the code artifacts
- ✎ Each item of functionality or function is identified
- ✎ Test data are devised to test each (lower-level) function separately
- ✎ Then, higher-level functions composed of these lower-level functions are tested

Regression Testing

- Each time a new module is added as part of integration testing, the software changes and needs to be re-tested.
- 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.
- In broader context, regression testing ensures that changes (enhancement or correction) do not introduce unintended new errors.
- When used?
 1. After integration testing
 2. Testing after module modification

∞ Measure the system's performance

- Running times of various tasks
- Memory usage, including memory leaks
- Network usage (Does it consume too much bandwidth? Does it open too many connections?)
- Disk usage (Does it clean up temporary files properly?)
- Process/thread priorities (Does it run well with other applications, or does it block the whole machine?)

Volume Testing

- ∞ Volume testing: System behavior when handling large amounts of data (e.g. large files)
 - Test what happens if large amounts of data are handled
- ∞ Load testing: System behavior under increasing loads (e.g. number of users)
- ∞ Test the system at the limits of normal use
 - Test every limit on the program's behavior defined in the requirements
 - Maximum number of concurrent users or connections
 - Maximum number of open files
 - Maximum request or file size

Stress Testing

- ☞ Stress testing: System behavior when it is overloaded
- ☞ Test the limits of system (maximum # of users, peak demands, extended operation hours)
- ☞ Test the system under extreme conditions
 - Create test cases that demand resources in abnormal quantity, frequency, or volume
 - Low memory
 - Disk faults (read/write failures, full disk, file corruption, etc.)
 - Network faults
 - Power failure
 - Unusually high number of requests
 - Unusually large requests or files
 - Unusually high data rates
- ☞ Even if the system doesn't need to work in such extreme conditions, stress testing is an excellent way to find bugs

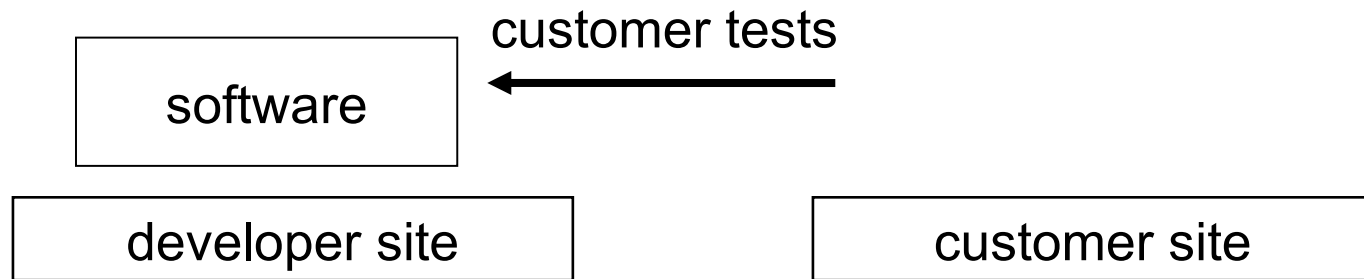
Security Testing

- ✎ Try to violate security requirements
- ✎ Any system that manages sensitive information or performs sensitive functions may become a target for illegal intrusion
- ✎ How easy is it to break into the system?
- ✎ Password usage
- ✎ Security against unauthorized access
- ✎ Protection of data
- ✎ Encryption

Acceptance Testing

- ✎ The client determines whether the product satisfies its specifications
- ✎ Acceptance testing is performed by
 - The client organization, or
 - The SQA team in the presence of client representatives, or
 - An independent SQA team hired by the client
- ✎ The key difference between product testing and acceptance testing is
 - Acceptance testing is performed on actual data
 - Product testing is performed on test data, which can never be real, by definition

Alpha Test



Beta Test



Concerns of Web-based Systems:

- Browser compatibility
- Functional correctness
- Usability
- Security
- Reliability
- Performance
- Recoverability

Examples of Test Automation Tools

Vendor	Tool
Hewlet Packard	QuickTest
IBM	Rational Functional Tester
Selenium	Selenium
Microfocus	SilkTest
AutomatedQA	TestComplete

This week we present

∞ Good Software Implementation Principles

- Variable naming, commenting, debugging, etc.

∞ Low Level Testing

- Applying Unit Testing
- The concept of drivers and stubs
- How to integrate the system at overall

∞ Testing Strategies and Approaches

- White-box testing and black-box testing
- Input domain partitioning
- Higher Level Testing