SOFTWARE ENGINEERING

Week 12 & 13
Software Testing

Agenda

1. Phases of Testing
2. Unit Testing
   1. White-Box
   2. Black-Box
3. Testing Strategies
   1. Top-down
   2. Bottom-up
4. Special Tests
5. Web-based Testing

Software Testing

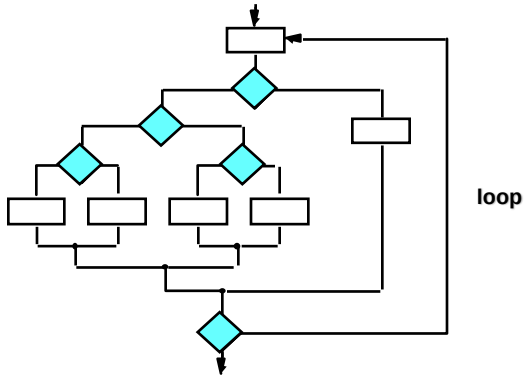
• Testing is the process of executing a program to find errors.

• Testing is planned by defining “Test Cases” in a systematic way.

• **A test case is a collection of input data and expected output.**

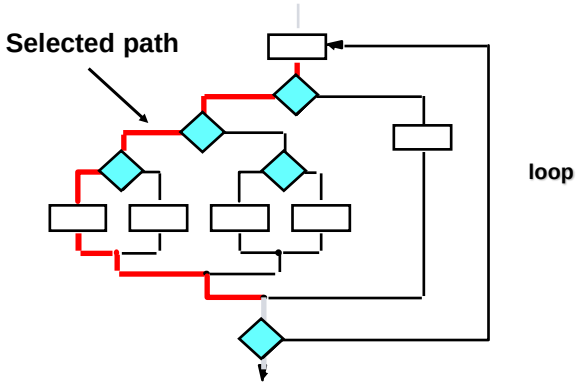
Exhaustive Testing

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



Selective Testing

- Instead of exhausting testing, a selective testing is more feasible.



5

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?

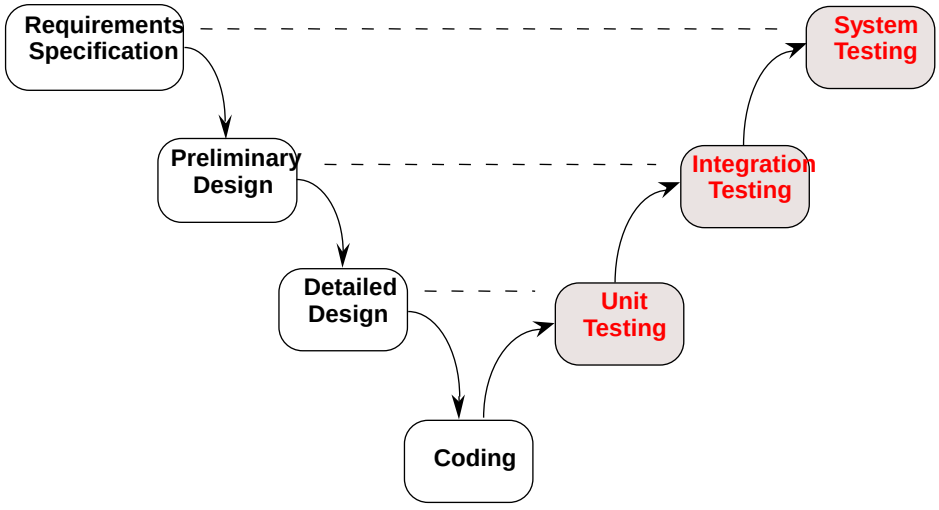
3. System Testing

Does it work within the overall system?

Does the program satisfy the requirements ?

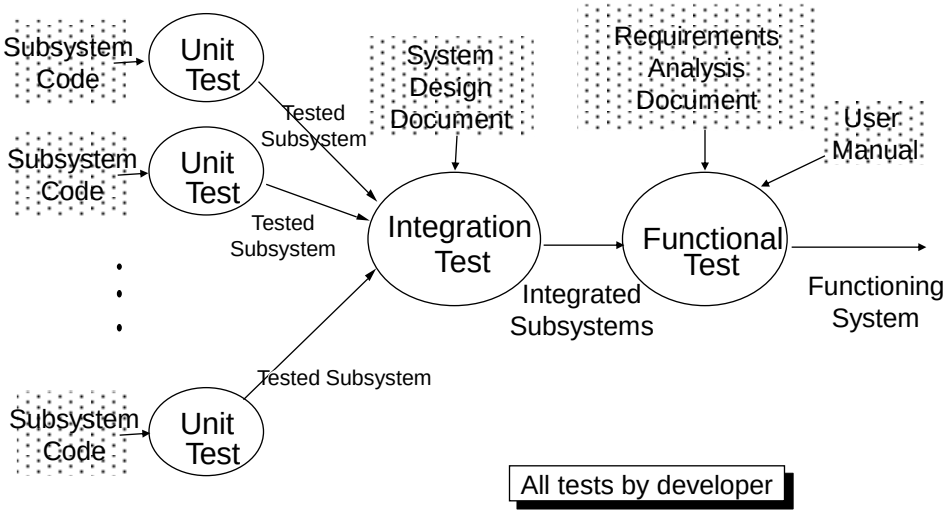
6

Levels of Testing



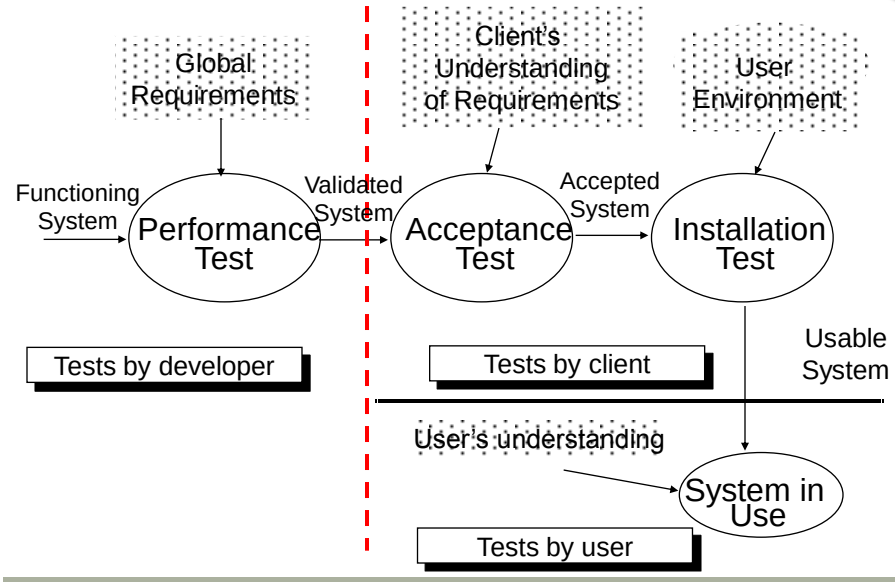
7

Testing Activities (1)



8

Testing Activities (2)

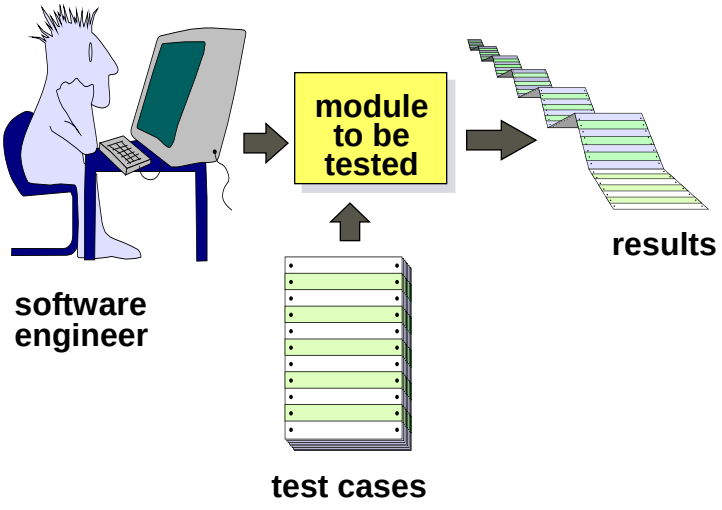


UNIT TESTING

Unit Testing

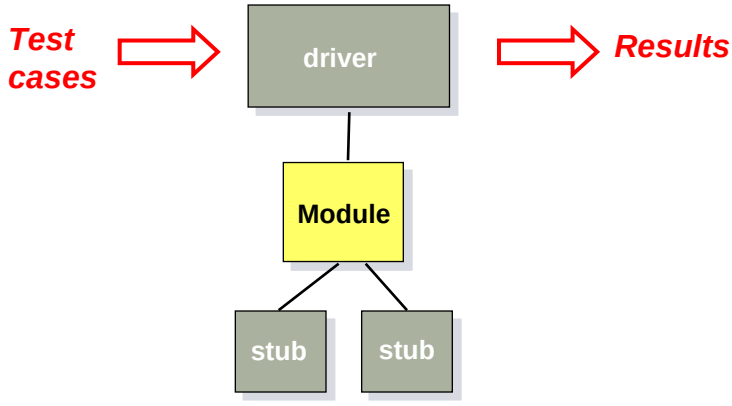
- Static Analysis:
  - Hand execution (Reading the source code)
  - Code inspection (Walk-through)
  - Automated tools checking for syntactic and semantic errors
- Dynamic Analysis:
  - Black-box testing (Test the **input/output** behavior)
  - White-box testing (Test the internal **logic**)

Unit Testing



Unit Test Environment

- Stubs and drivers should be written in unit testing.



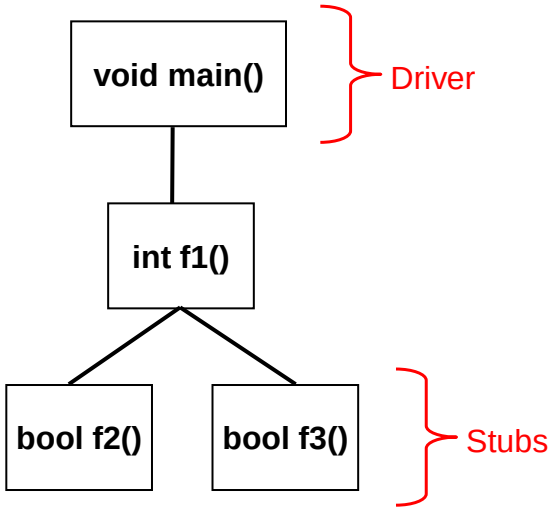
13

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.
- A driver is a caller module, which calls its stubs:
  - Once or several times,
  - Each time checking the value returned.

14

Example: Stubs and driver (1)



15

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);
}
```

16

Unit Testing Methods

- 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

17

Verification and Validation

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

- White box testing is used for verification since it is done at the level of the implementation to ensure that the code performs correctly.

- Black box testing is used for validation since the tests are done at the interface level and can therefore test the requirements. It can also be used for verification in unit testing to check that the implementation satisfies the design.

18

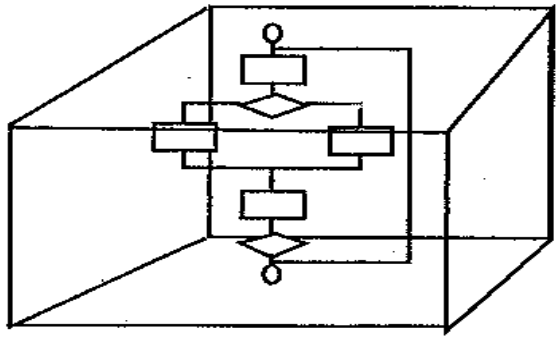
White Box Testing

White-Box Testing (1)

- White box testing is primarily used during unit testing
- Unit testing is usually performed by the programmer who wrote the code
- In rare cases, an independent tester might do unit testing on your code
- **Code coverage:** At a minimum, every line of code should be executed by at least one test case.

White-Box Testing (2)

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



White-Box Testing (3)

White-box testing is also known as Basis Path Testing Method.

1. Guarantee that all independent paths within a module have been executed at least once.
2. Execute all logical decisions on their *true* and *false* sides.
3. Execute all loops at their boundaries and within their operational bounds.
4. Use internal data structures to assure their validity.

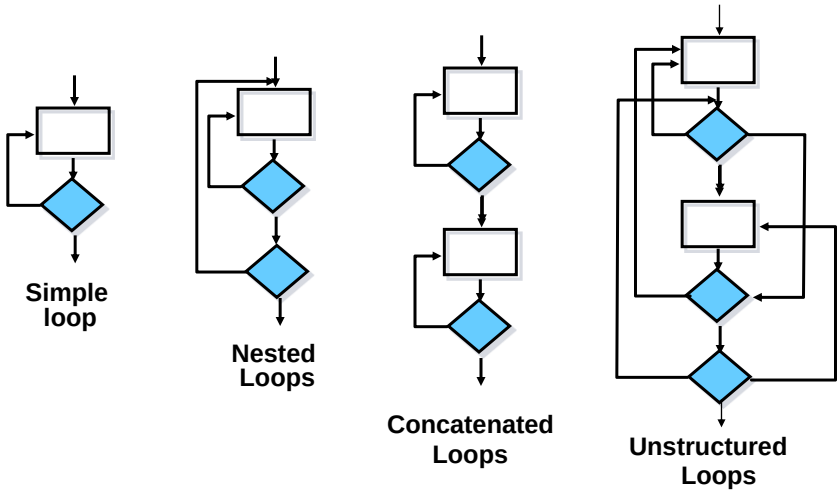
Flow graph for White box testing

- To help the programmer to 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 statements and branches; the objective can be to:
    - cover all possible paths (often infeasible)
    - cover all possible nodes (simpler)
    - cover all possible edges (most efficient)

Branch Coverage

- If statements
  - test both positive and negative directions
- Switch statements
  - test every branch
  - If no default case, test a value that doesn't match any case
- Loop statements
  - test for both 0 and > 0 iterations

Loop Testing (1)

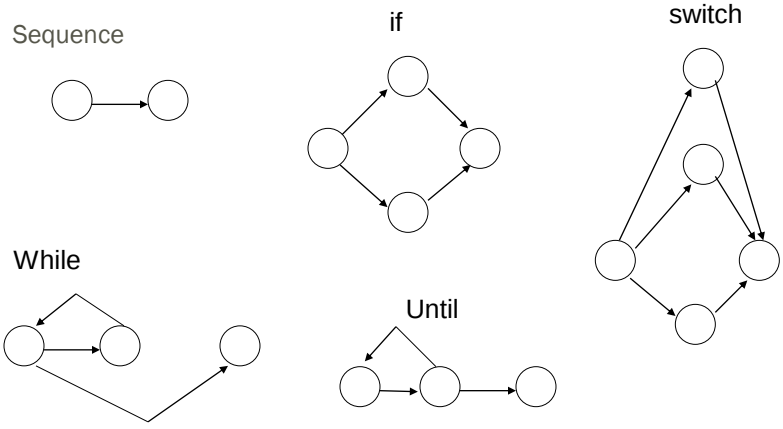


Loop Testing (2)

- Design test cases based on looping structure of the routine
- Testing loops
  - Skip loop entirely
  - Exactly one pass
  - Exactly two passes
  - N-1, N, and N+1 passes
    where N is the maximum number of passes
  - M passes, where 2 < M < N-1

Flow Graph Notation

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



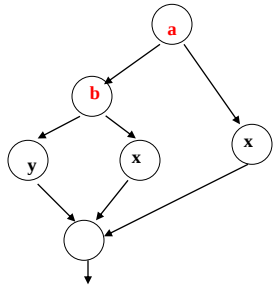
Compound Logic

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

then procedure **x**

else procedure **y**

ENDIF



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

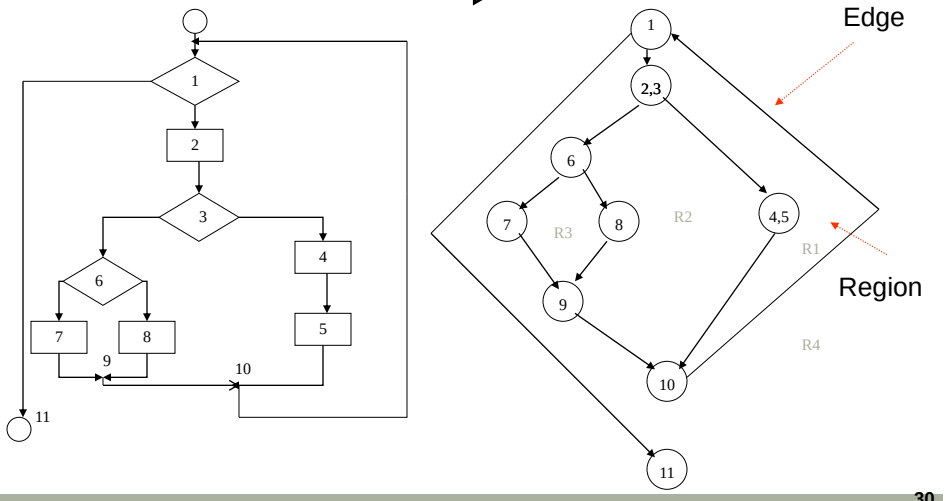
- **Another method:**
  V(G) = P + 1

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

29

Example-1

Flow chart ⇒ **Flow graph**



30

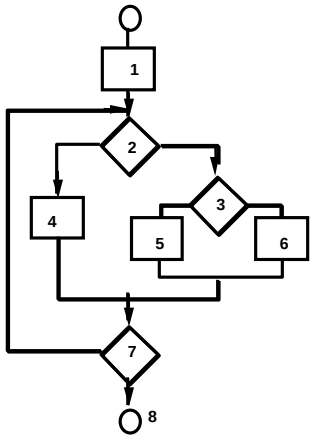
Example-1 (Cyclomatic Complexity)

- V(G) = E – N + 2
  = 11 – 9 + 2 = 4

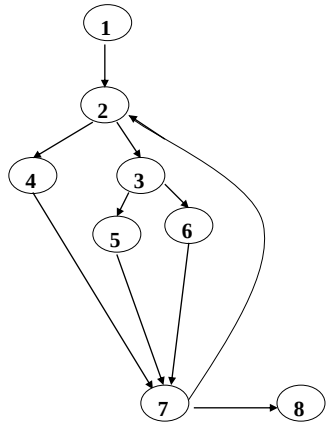
- Other method:
  V(G) = P + 1
  = 3 + 1 = 4

Example-2

Flow chart



Flow graph



Example-2 (Cyclomatic Complexity)

First, we calculate V(G):
V(G) = E – N + 2
= 10 – 8 + 2 = 4

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

Test Cases

Test cases should cover followings:

- interface
- local data structures
- boundary conditions
- independent paths
- error handling paths

Tests Cases for Paths

• After determining independent paths,, we derive "Test Cases" to exercise these paths.

Test Case for Path1:
Inputs:
variable1 = data1
variable2 = data2
...
Expected results: Correct or incorrect values

Test Case for Path2:
Inputs:
variable1 = data3
variable2 = data4
...
Expected results: Correct or incorrect values

35

Example: Script for Unit Testing

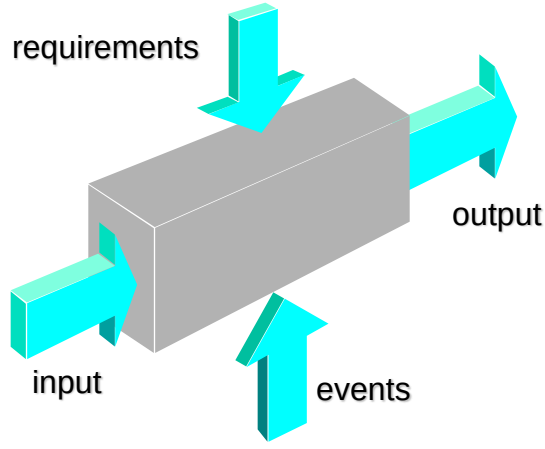
- This is the format for a test plan to show what you're planning to do.
- It should to be filled to show what happened when you run tests.

Scenario # : 1	Tester: AAA BBB	Date of Test: 01/01/2010		
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				

36

Black Box Testing

Black-Box Testing

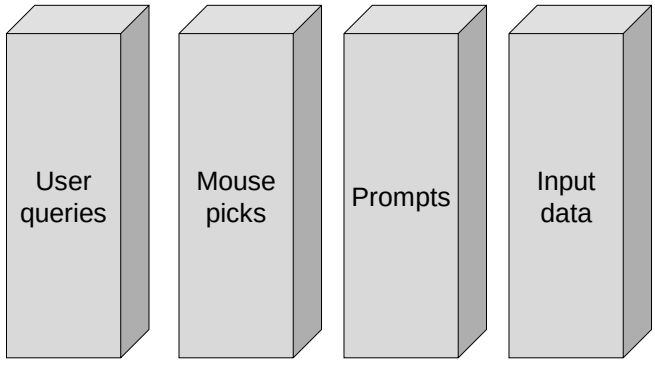


Black Box Testing

- 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

39

Equivalence Partitioning for Input Domain



40

Examples: Input Data

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

Domain of Inputs

- Individual input values
- Combinations of inputs
  - Individual inputs are not independent from each other
  - Programs process multiple input values together, not just one at a time
  - Try many different combinations of inputs in order to achieve good coverage of the input domain
  - Ordering of inputs: In addition to the particular combination of input values chosen, the ordering of the inputs can also make a difference

Examples of Input Domain

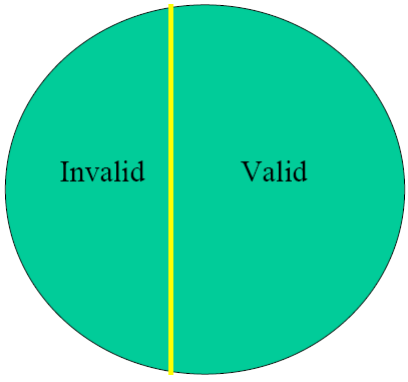
- Boolean value
  - True or False
- Numeric value in a particular range
  - 99 <= N <= 999
  - Integer, Floating point
- One of a fixed set of enumerated values
  - {Jan, Feb, Mar, ...}
  - {VisaCard, MasterCard, DiscoverCard, ...}
- Formatted strings
  - Phone numbers
  - File names
  - URLs
  - Credit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

Domain of Outputs

- In addition to covering the input domain, make sure your tests thoroughly cover the output domain
- What are the valid output values?
- Is it possible to select inputs that produce invalid outputs?

Equivalence Partitioning (1)

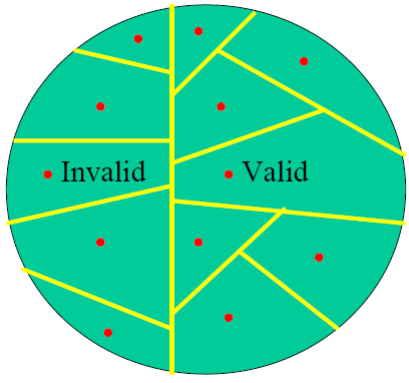
- First-level partitioning: Valid vs. Invalid input values



45

Equivalence Partitioning (2)

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



46

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	?	?

47

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	555-5555 (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 *Similar for Prefix and Suffix* Invalid format 5555555, (555)(555)5555, etc.

48

Boundary Value Analysis (1)

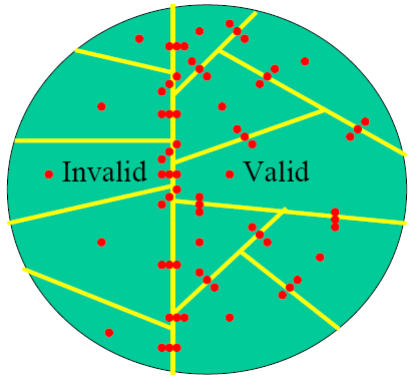
- Many errors tend to occur at the boundaries of the input domain.
- If an input condition specifies range bounded by values a (minimum) and b (maximum), test cases should be designed with values a and b, just above and just below a and b, respectively.
- If internal data structures have boundaries (e.g., an array has a defined limit of 100 entries) be certain to design a test case to exercise the data structure beyond its boundary.

Boundary Value Analysis (2)

- 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 <= N <= 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 <= N <= 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	
(87.3)		x			87.3	crashes!
(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. (It can be used to get black/white of an image.)
- Normalization Algorithm:
  - First, the array of N elements is searched for the smallest and largest non-negative elements.
  - Secondly, the range is calculated as max - 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

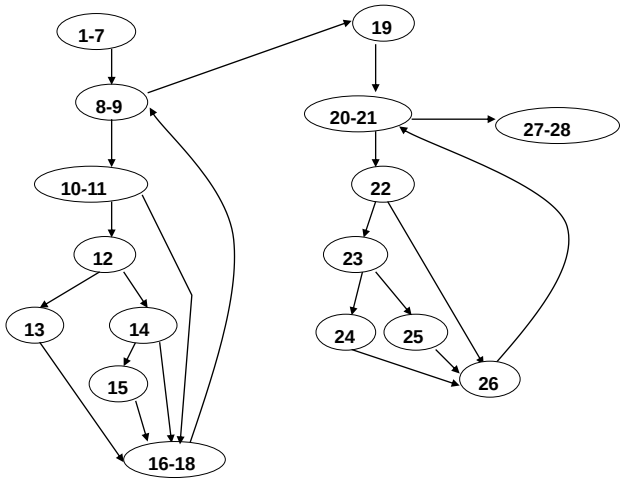
```
1. int normalize (int A[], int N)
2. {
3. int range, max, min, i, valid;
4. range = 0;
5. max = -1;
6. min = -1;
7. valid = 0;
8. for (i = 0; i < N; i++)
9. {
10.  if (A[i] >= 0)
11.  {
12.    if (A[i] > max)
13.       max = A[i];
14.    else if (A[i] < min)
15.            min = A[i];
16.    valid++;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

65

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)

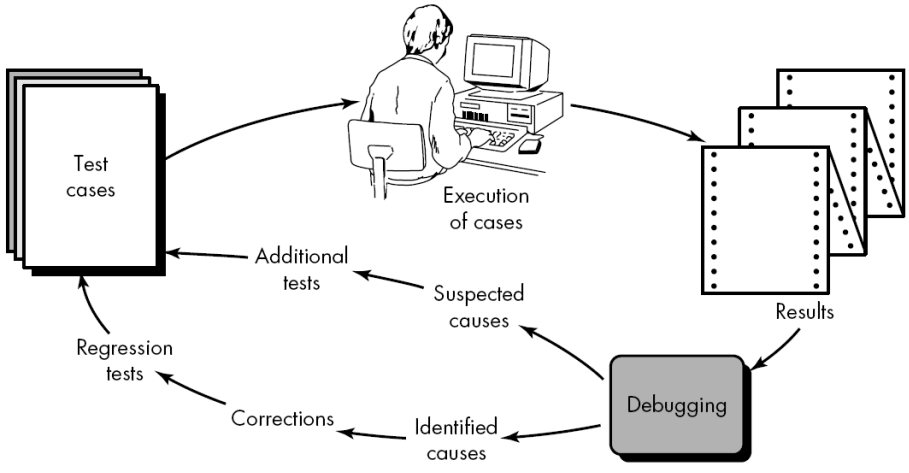
Input Condition: N=5, A[ ] = {10, 10, 10, 10, 10}

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

66

Debugging

Debugging Process



Error Fixing

Error fixing is two steps:

1. Locating the error:

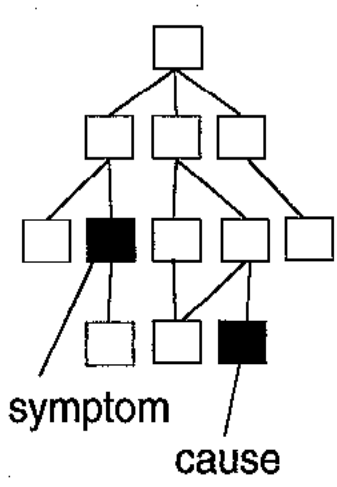
Time required to determine the nature and location of the error is usually %80 of debugging time.

2. Correcting the error:

Time required to correct the error is usually %20 of debugging time.

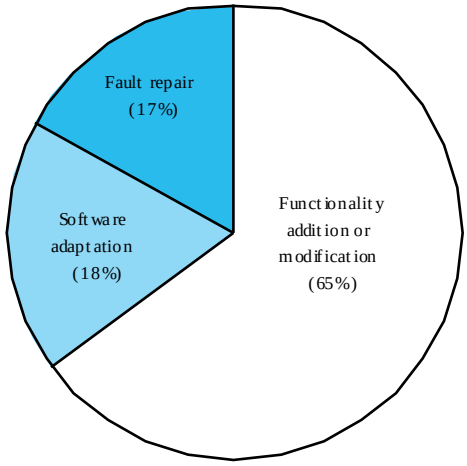
Symptoms and Causes of Bugs

- symptom and cause may be in different locations
- cause may be due to a combination of errors
- cause may be due to a system error



Software Maintanence

- Correction of defects (bugs, errors)
- Adaptation to a new version of Operating System, DBMS, or hardware
- Enhancement (adding new functions)



Examples of Test Automation Tools

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

INTEGRATION TESTING STRATEGIES

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.
- Testing should be conducted by the developer of the software and also by an independent test group.

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.

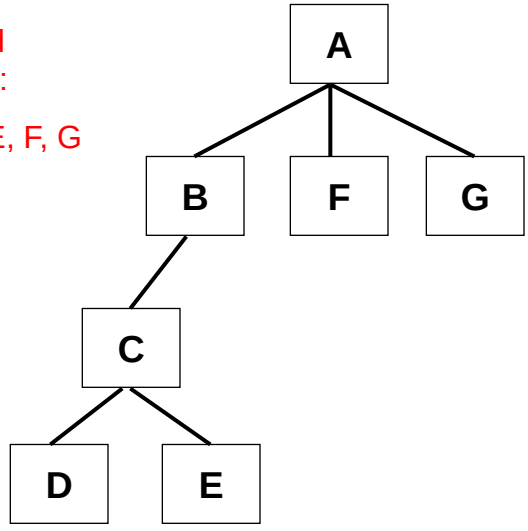
• After each integration step, a regression test is conducted.

75

Big bang integration testing

Integration and testing at once:

1. A, B, C, D, E, F, G



76

Top-down Testing Strategy

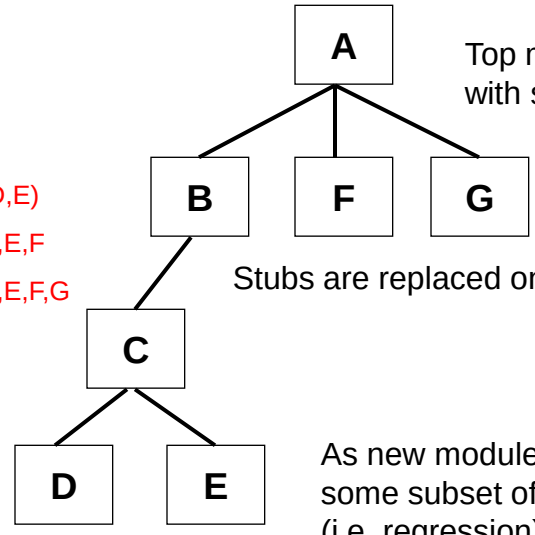
- First, test the top layer or the controlling subsystem
- Then combine all the subsystems that are called by the tested subsystems and test the resulting collection of subsystems
- Do this until all subsystems are incorporated into the test
- Special routine is needed to do the testing, ***Test stub*** :
  - A routine that simulates the activity of a missing subsystem by answering to the calling sequence and returning back fake data.

77

Example1: Top-Down Integration (Depth-first)

STEPS:

1. A
2. A,B
3. A,B,C
4. A,B,C,(D,E)
5. A,B,C,D,E,F
6. A,B,C,D,E,F,G



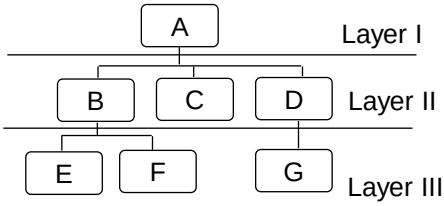
Top module is tested with stubs

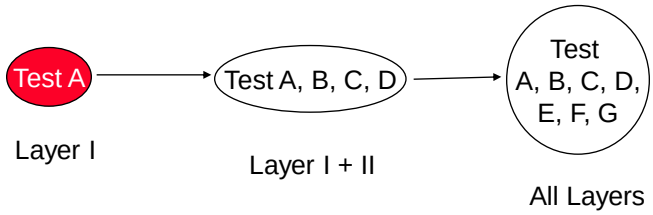
Stubs are replaced one at a time

As new modules are integrated, some subset of tests is re-run (i.e. regression)

78

Example2: Top-down Integration (Breadth-first)





79

Pros and Cons of top-down integration

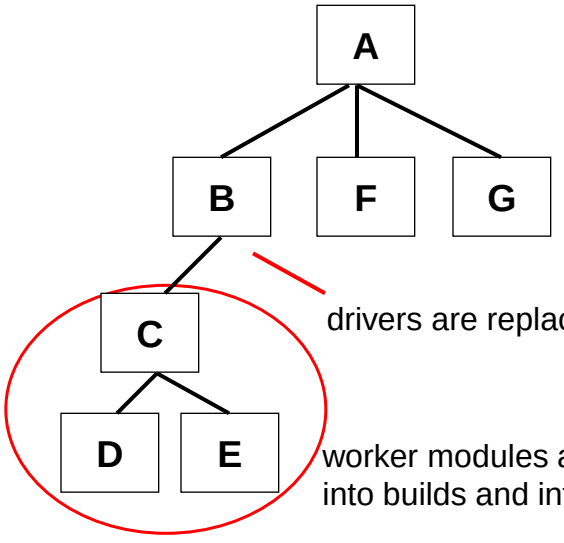
- Test cases can be defined in terms of the functionality of the system (functional requirements)
- Writing stubs can be difficult: Stubs must allow all possible conditions to be tested.
- Possibly a very large number of stubs may be required, especially if the lowest level of the system contains many methods.

80

Bottom-up Testing Strategy

- First, the subsystem in the lowest layer of the call hierarchy are tested individually
- Then the next subsystems are tested that call the previously tested subsystems
- This is done repeatedly until all subsystems are included in the testing
- Special routine needed to do the testing, ***Test Driver***:
  - A routine that calls a particular subsystem and passes a test case to it

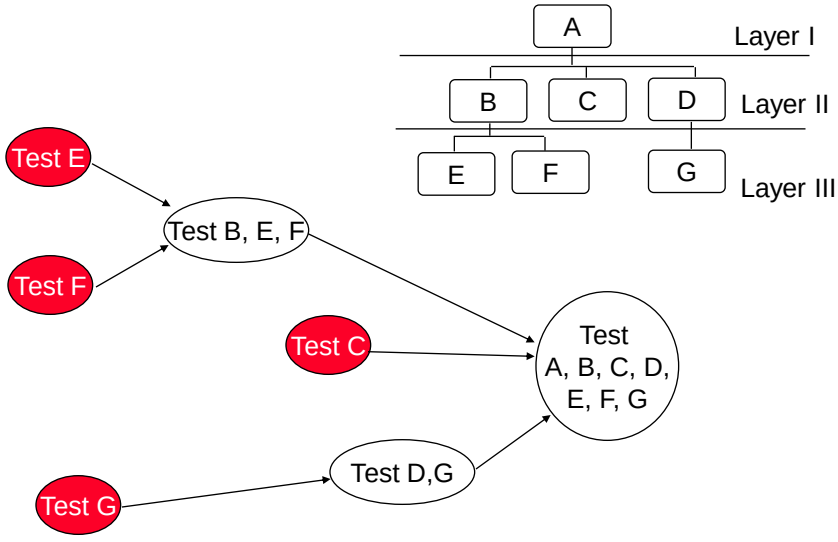
Example1: Bottom-Up Integration (Depth-first)



STEPS:

1. D,E
2. C,D,E
3. B,C,D,E
4. A,B,C,D,E
5. A,B,C,D,E,F
6. A,B,C,D,E,F,G

Example2: Bottom-up Integration (Breadth-first)



83

Pros and Cons of bottom-up integration testing

- Bad for functionally decomposed structured systems:
  - Tests the most important subsystem last
- Useful for integrating the following systems
  - Object-oriented systems
  - Real-time systems
  - Systems with strict performance requirements

84

Special Tests

Special Tests

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

System Testing

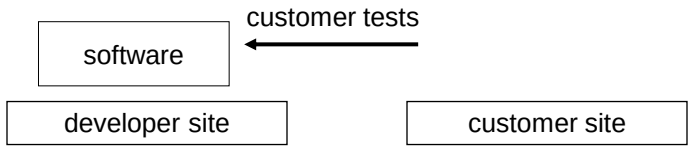
1. Functional Testing
2. Performance Testing
3. Acceptance Testing
4. Installation Testing

Regression Testing

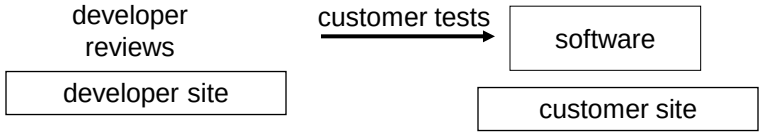
- Each time a new module is added as part of integration testing, the software changes and needs to re-tested.
- Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.
- In broader context, regression testing ensures that changes (enhancement or correction) do not introduce unintended new errors.
- When used?
  1. Integration testing
  2. Testing after module modification

Alpha Test & Beta Test

Alpha Test



Beta Test



89

Performance Testing

- 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?)

90

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

91

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

92

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

Usability Testing

- Test user interfaces.
- Evaluate response times and time to perform a function.
- Is the user interface intuitive, easy to use, organized, logical?
- Does it frustrate users?
- Are common tasks simple to do?
- Does it conform to platform-specific conventions?
- Get real users to sit down and use the software to perform some tasks
- Get their feedback on the user interface and any suggested improvements
- Report bugs for any problems encountered

Recovery Testing

- Test system's response to presence of errors or loss of data.
- Try turning the power off crashing the program at arbitrary points during its execution
  - Does the program come back up correctly when you restart it?
  - Was the program's persistent data corrupted (files, databases, etc.)?
  - Was the extent of user data loss within acceptable limits
- Can the program recover if its configuration files have been corrupted or deleted?

Testing Web-based Systems

Concerns of Web-based Systems:

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

Web-based Testing

- Security is one of the major risks of Internet applications. You must validate that the application and its data are protected from unauthorized access
- **Unit Testing:** Done at the object, component, HTML page, or applet level
- **Integration Testing:** Verify the passing of data and control between units or components; this includes testing the navigation, links, and data exchanges
- **System Testing:** Tests the Web application as a whole, and how it interacts with other Web applications or systems