SOFTWARE QUALITY Assurance & Test

LECTURE # 17

SOFTWARE TESTING - III (DYNAMIC)
WHITE BOX TESTING

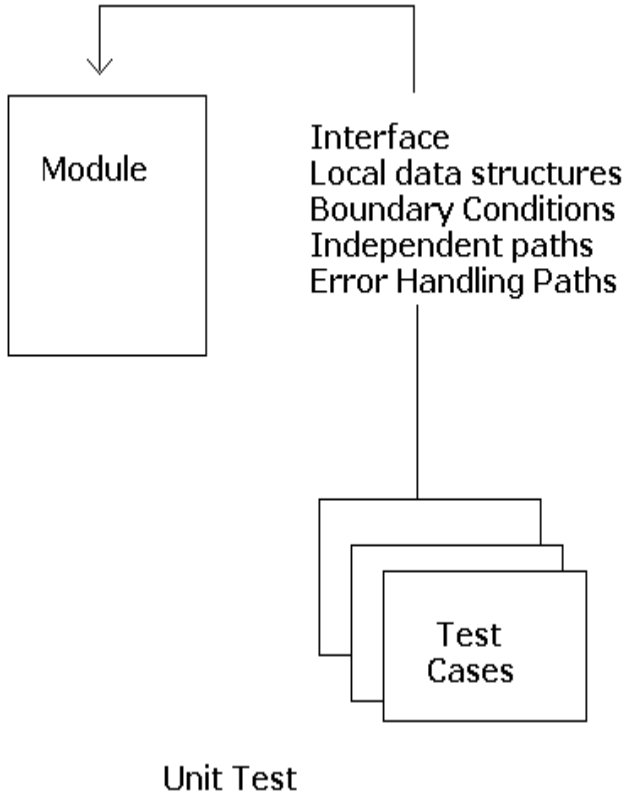
chenbo@etao.net

Topics to Cover

- Unit Testing
- Code Coverage and its Types
- Control Flow Graph
- Cyclomatic Complexity
- Graph matrices
- Basis Path Testing
- Control Flow Testing
- Condition Testing
- Loop Testing
- Data Flow Testing
- Data Flow Coverage Criteria
- Terms used in data flow testing
- Data Flow Coverage Concept
- DC Path

Unit Testing

- First level of testing. Refers to testing program units in isolation.
- A program unit implements a function, it is natural to test the unit before it is integrated with other units.
- Unit testing focuses on verification effort on the smallest unit of software design (the software component or module).
- Using the component level design description as a guide, important control paths are tested to uncover errors within the boundary of the module.
- The unit test is white box oriented and the step can be conducted in parallel for multiple components.

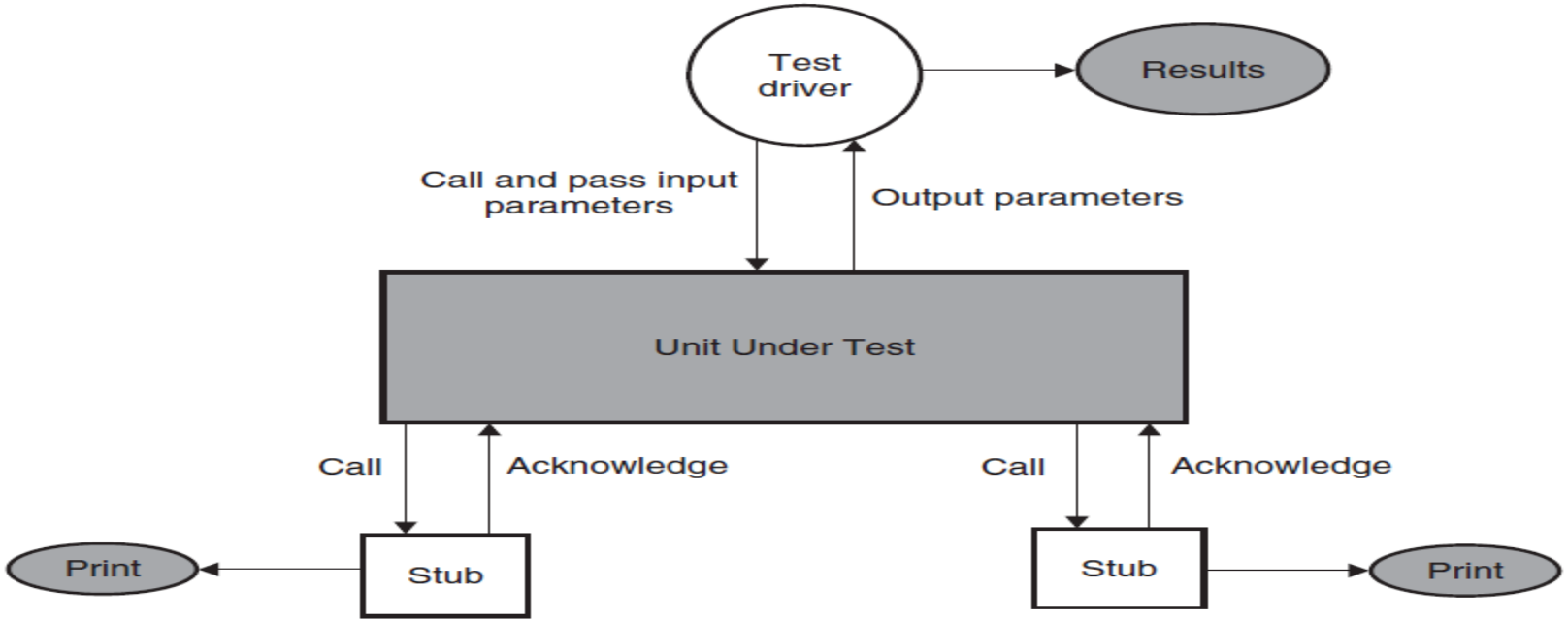


Dynamic Unit Testing

- Dynamic unit testing is execution based testing
- As your programs become more complicated, and the number of functions increases, you will need to test each function separately, therefore in this testing, a program unit is actually executed in isolation.
- Losing strategy: Write each function and execute them all together.
  - It is difficult to debug all the functions at once
  - Multiple errors interact
- Winning strategy: Test each function separately.
  - Make sure each function works before you test it with other functions.
  - In the long run, this saves testing and debugging time.
- How can you test a function that depends on other functions?

Dynamic Unit Testing Environment

- An environment for dynamic unit testing is created by emulating the context of the unit under test
- The caller unit is known as a *test driver, and all the emulations of the units* called by the unit under test are called *stubs*



Test Driver and Stubs

- **Test Driver** *is a software module used to invoke a module under test and, often, provide test inputs, control and monitor execution, and report test results* (IEEE, 1990)
- A piece of code that passes test cases to another piece of code.
- The unit under test executes with input values received from the driver and, upon termination, returns a value to the driver.
- The driver compares the actual outcome, that is, the actual value returned by the unit under test with the expected outcome from the unit and reports the ensuing test result.
- For example, if you wanted to move a Player instance,Player1, two spaces on the board, the driver code would be

```
movePlayer(Player1, 2);
```

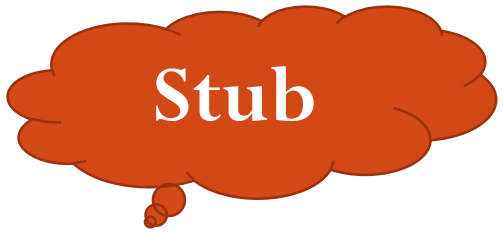
- This driver code would likely be called from the main method.

Test Driver and Stubs

- **Stub** is a "dummy subprogram" that replaces a unit that is called by the unit under test.
- A piece of code that simulates the activity of missing components.
- If the function A you are testing calls another function B, then use a simplified version of function B, called a stub.
- A stub returns a value that is sufficient for testing.
- The stub does not need to perform the real calculation.
- A stub performs two tasks.
  - First, it shows an evidence that the stub was, in fact, called. Such evidence can be shown by merely printing a message.
  - Second, the stub returns a pre-computed value to the caller so that the unit under test can continue its execution.

Null Case Testing

```
void function_under_test(int& x, int& y) {
...
p = price(x);
...
}
```

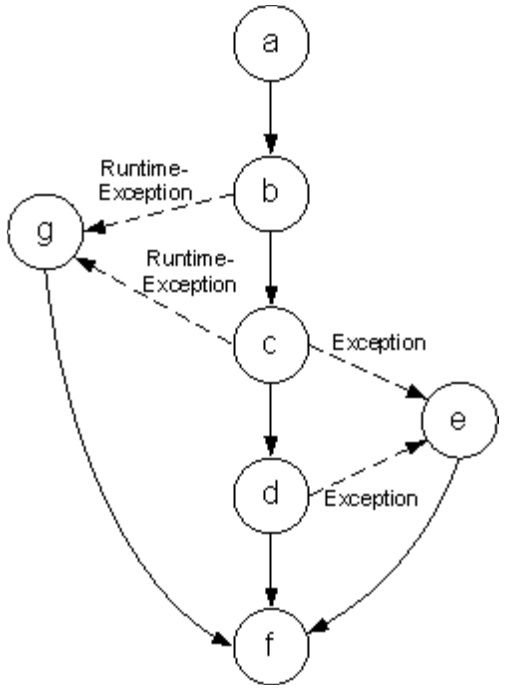



```
double price(int x) {return 10.00;}
```

- The value returned by function price is good enough for testing.
- The real price() function may not yet have been tested, or even written.
- **Stubs and drivers are often viewed as throwaway code (Kaner, Falk et al., 1999). However, they do not have to be thrown away: Stubs can be "filled in" to form the actual method. Drivers can become automated test cases.**

Control Flow Testing

- This testing approach identifies the execution paths through a module of program code and then creates and executes test cases to cover those paths.
- Pick enough paths to assure that every source statement is executed at least once.
- Path: A sequence of statement execution that begins at an entry and ends at an exit.

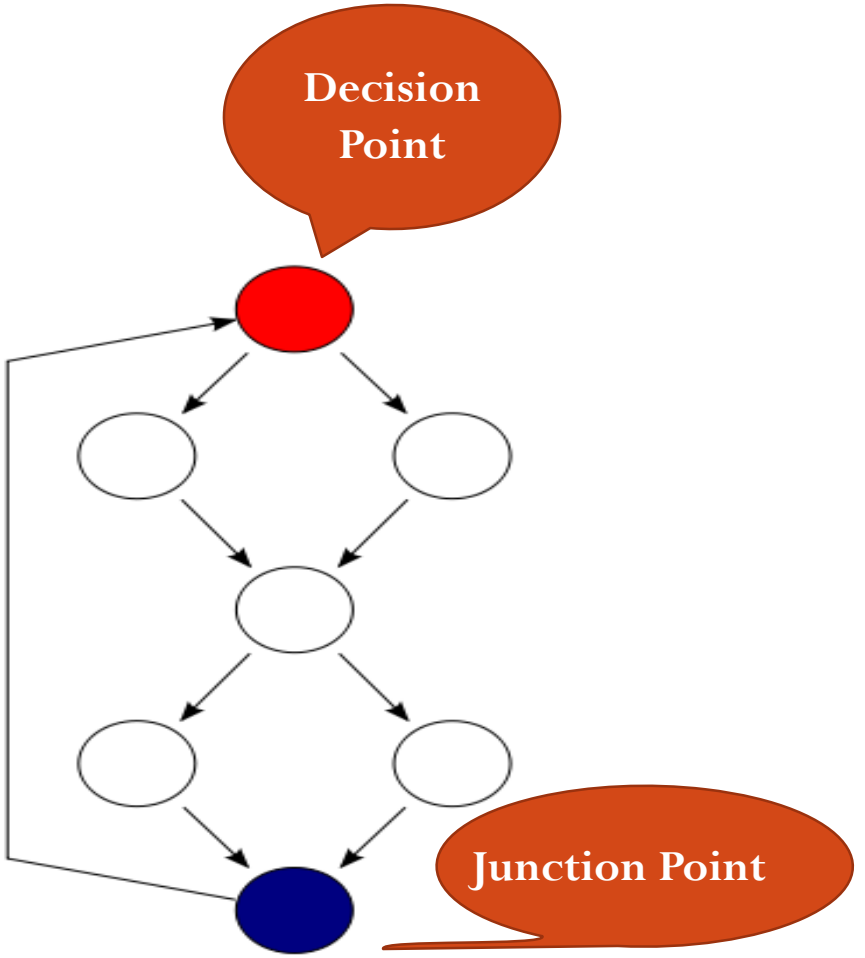


Control Flow Testing

- A simple notation for the representation of control flow is called a flow graph (or program graph).
- In flow graph each circle (a flow graph node) represents one or more procedural statements.
- A sequence of process boxes and a decision diamond can map into a single node.
- The arrows on the flow graph called edges or links represents flow of control.
- An edge must terminate at a node even if the node does not represent any procedural statements.
- Areas bounded by edges and nodes are called regions.
- A compound condition occurs when one or more Boolean operators is present in a conditional statement
- Each node that contains a condition is called a predicate node & is characterized by two or more edges originating from it.

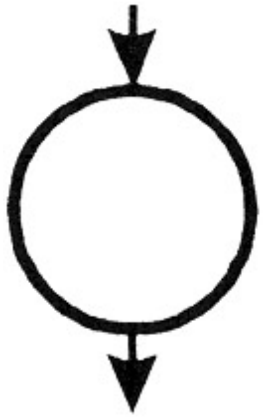
Elements of Control Flow Graph

- Process Block
- Decision Point
- Junction Point



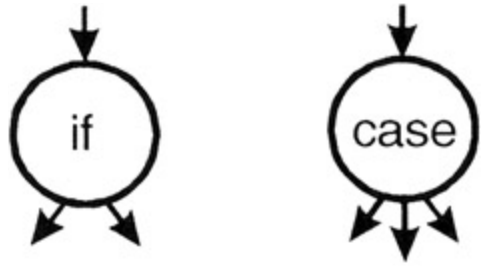
Process Block

- A process block is a sequence of program statements that execute sequentially from beginning to end.
- Once the block is initiated, every statement within it will be executed sequentially.
- Process blocks are represented in control flow graphs by a bubble with one or more entries and one exit.



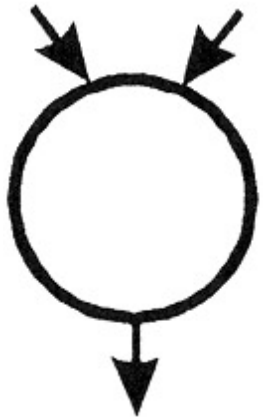
Decision Point

- A decision point is a point in the module at which the control flow can change.
- Most decision points are binary and are implemented by if-then-else statements.
- Multi-way decision points are implemented by case statements.
- They are represented by a bubble with one entry and multiple exits.



Junction Point

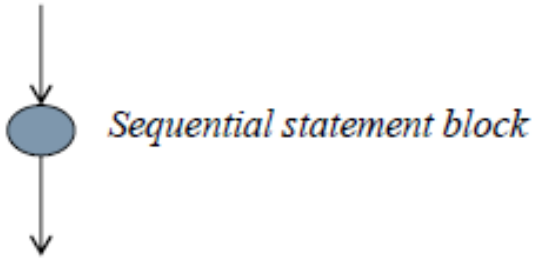
- A junction point is a point at which control flows join together.



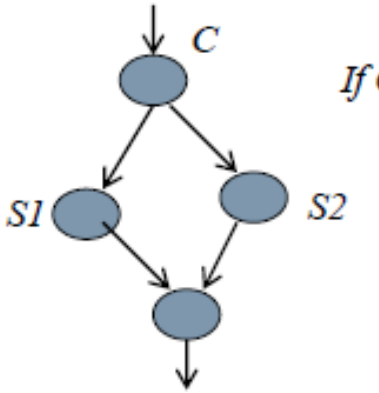
Path

- A path through a program is a sequence of statements that starts at an entry, junction, or decision and ends at another (possible the same), junction, decision, or exit.
- A path may go through several junctions, processes, or decisions, one or more times.
- Paths consist of segments.
- The length of a path is the number of links in a path.

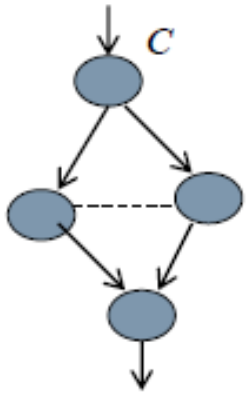
Flow Graphs of various blocks



Sequential statement block

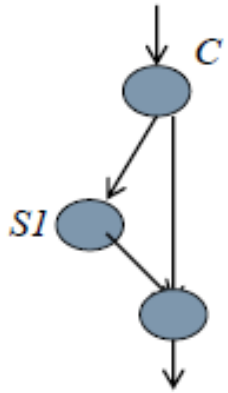


If C Then S1 else S2;



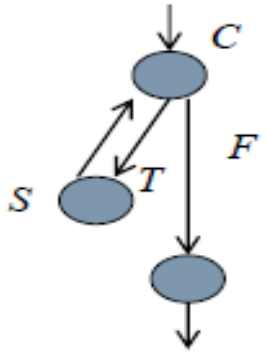
Case C of

L1: S1;
L2: S2;
...
Kn: Sn;
end;

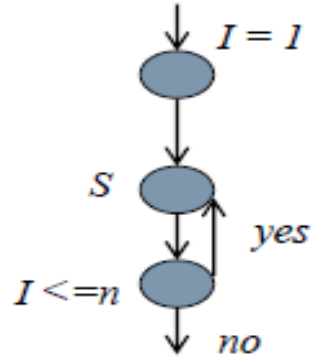


If C Then S1;

Flow Graphs of various blocks

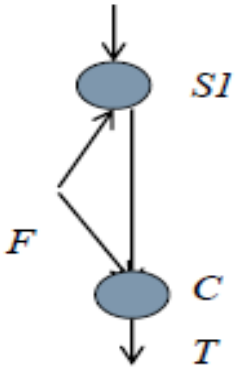


While C do S;



For loop:

for I = 1 to n do S;

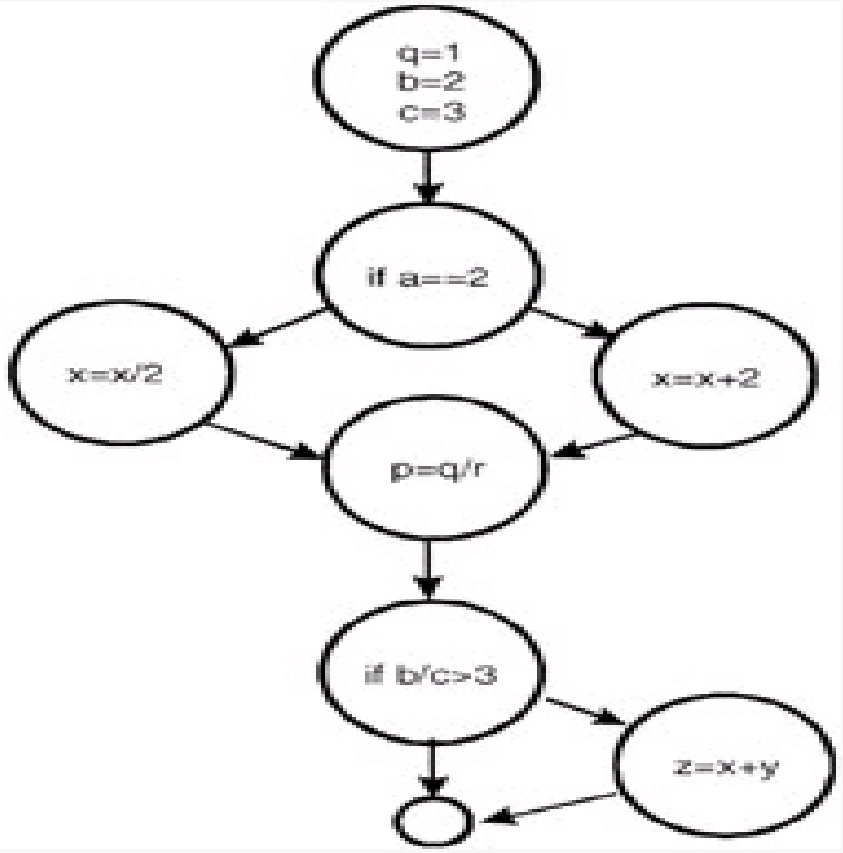


Do loop:

do S1 until C;

Flow Graph

```
q=1;
b=2;
c=3;
if (a==2) {x=x+2;}
else {x=x/2;}
p=q/r;
if (b/c>3) {z=x+y;}
```



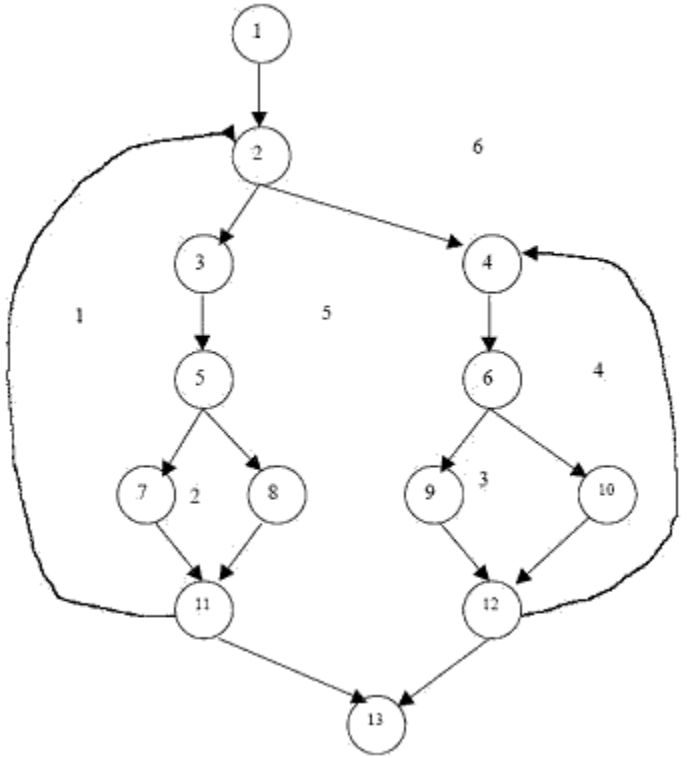
Cyclomatic Complexity

- Cyclomatic complexity is a software metric that provides a quantitative measure of the global complexity of a program.
- When this metric is used in the context of the basis path testing, the value computed for cyclomatic complexity defines the number of independent paths in the basis set of a program.
- Three ways to compute cyclomatic complexity:
  - The number of regions of the flow graph correspond to the cyclomatic complexity.
  - Cyclomatic complexity, V(G), for a flow graph G is defined as V(G) = E - N + 2, where E is the number of flow graph edges and N is the number of flow graph nodes.
  - Cyclomatic complexity, V(G) = P + 1

    where P is the number of predicate nodes contained in the flow graph G.

Example

- Region, R= 6
- Number of Nodes = 13
- Number of edges = 17
- Number of Predicate Nodes = 5
- Cyclomatic Complexity, V(C) :
  - V(C) = R = 6; Or
  - V(C) = Predicate Nodes + 1 =5+1 =6 Or
  - V(C)= E-N+2 = 17-13+2



Example (Home Assignment)

```
i=1;
total.input = total.valid = 0;
Sum=0;
                Do
                increment total.input by 1;
                IF value[i] >= minimum and value[i] <=maximum
                                THEN increment total.valid by 1;
                                                sum = sum + value[I];
                ENDIF
                increment i by 1;
while value[i]<>-999 and total.input <= 100
end
                IF value[i] >0
                                THEN average = sum / total.valid;
                                ELSE average = -999;
                ENDIF
END average
```

C(G), Quality and Testability

- C(G) < 5

  'simple and easy to understand'

- C(G) ≤ 10

  'not too difficult'

- C(G) > 20

  'complexity perceived as high'

- C(G) > 50

  'for all practical purposes untestable'

- However, no discovered relationship between C(G) and "bugginess"

Basic path testing

- First proposed by Tom McCabe.
- Used as a guide for defining a basis set of execution path.
- Guarantee to execute every statement in the program at least one time.
- *Basis path testing (McCabe, 1976) is a means for ensuring that all independent paths through* a code module have been tested.
- An *independent path is any path through the code that* introduces at least one new set of processing statements or a new condition. (Pressman, 2001)
- Basis path testing provides a minimum, lower-bound on the number of test cases that need to be written.

Basic path testing- Steps

- Step 1 : Using the design or code as a foundation, draw a corresponding flow graph.
- Step 2: Determine the cyclomatic complexity of the resultant flow graph.
- Step 3: Determine a basis set of linearly independent paths.
- For example,

  path 1: 1-2-4-5-6-7

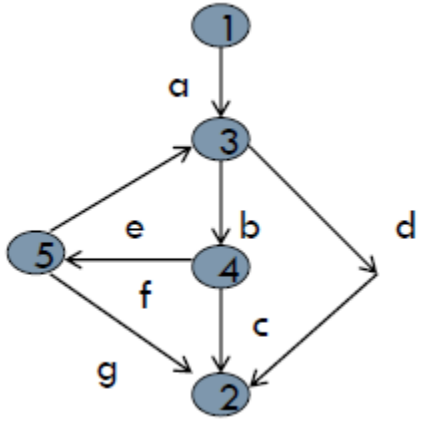
  path 2: 1-2-4-7

  path 3: 1-2-3-2-4-5-6-7

  path 4: 1-2-4-5-6-5-6-7
- Step 4: Prepare test cases that will force execution of each path in the basis set.
- Step 5: Run the test cases and check their results

Graph Matrices

- Graph Matrix is a data structure that assists in basis path testing.
- A graph matrix is a square matrix whose size is equal to the number of nodes on the flow graph. Each row and column corresponds to an identified node, and matrix entries corresponds to connections between nodes.
- Referring to the figure each node on the flow graph is identified by numbers, while each edge is identified by the letters. A letter entry is made in the matrix to correspond to a connection between two nodes. For example node 3 is connected to node 4 by edge b.



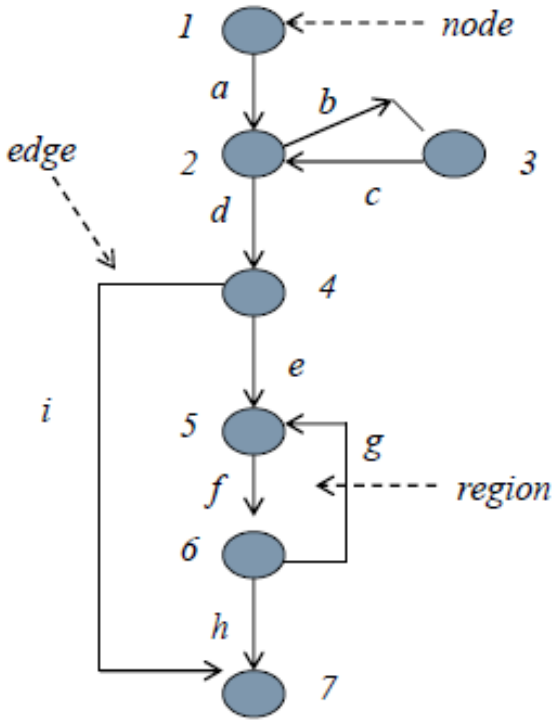
Graph Matrices

- Graph Matrix is nothing more than a tabular representation of a flow graph. However by adding a link weight to each matrix entry, the graph matrix can become a powerful tool for evaluating program control structure during testing.
- The link weight 1(a connection exists) and 0(a connection does not exists).

NODE	1	2	3	4	5
1			a		
2					
3		d		b	
4		c			f
5		g	e		

Graph Matrix

Graph Matrices



	1	2	3	4	5	6	7
1		a					
2			b	d			
3		c					
4					e		i
5						f	
6					g		h
7							

	1	2	3	4	5	6	7	
1		1						1 - 1 = 0
2			1	1				2 - 1 =1
3		1						1 -1 = 0
4					1		1	2 - 1 =1
5						1		1 -1 = 0
6					1		1	2 -1 = 1
7								

---

3 +1 = 4

White-Box Software Testing Methods

- Code Coverage is defined by the following types.
    1. Statement Coverage
    2. Segment Coverage
    3. Method Coverage
    4. Branch Coverage
    5. Compound Condition Coverage
    6. Basis Path Testing
    7. Domain Testing
    8. Data Flow Testing
    9. Loop Testing

Code Coverage

- Code coverage is a measure used in software testing to describe the degree to which the source code of a program has been tested.
- An example of what code coverage statistics can mean is that if there is a method with 100 lines of code, and only 75 of these lines are actually executed when tests are being run, then the method is considered to have a code coverage of 75 percent.

```
  :     /**
  :      * Sets the bank account's balance.
  :      *
  :      * @param  float $balance
  :      * @throws InvalidArgumentException
  :      * @access public
  :      */
  :     public function setBalance($balance)
  :     {
2 :         if ($balance >= 0) {
1 :             $this->balance = $balance;
1 :         } else {
1 :             throw new InvalidArgumentException;
  :         }
1 :     }
```

Method Coverage

- Method coverage is a measure of the percentage of methods that have been executed by test cases. Undoubtedly, your tests should call 100% of your methods.
- Method coverage of 50%, for example, means that half of the methods have been called

```
1    int foo (int a, int b, int c, int d, float e)  {
2          float e;
3          if (a == 0)  {
4               return 0;
5          }
6          int x = 0;
7          if ((a==b) OR ((c == d) AND bug(a) )) {
8               x=1;
9          }
10          e = 1/x;
11    return e;
12     }
```

Method Coverage

- In the code shown in previous slide, we attain 100% method coverage by calling the foo method.
- **Consider Test Case 1: the method call foo(0, 0, 0, 0, 0.), expected return value of 0.**
- If you look at the code, you see that if **'a'** has a value of 0, it does not matter what the values of the other parameters are – so we will make it really easy and make them all 0. Through this one call we attain 100% method coverage.

Statement Coverage

- Statement coverage is a measure of the percentage of statements that have been executed by test cases. Your objective should be to achieve 100% statement coverage through your testing.
- In Test Case 1, we executed the program statements on lines 1-5 out of 12 lines of code. As a result, we had 42% (5/12) statement coverage from Test Case 1. We can attain 100% statement coverage by one additional test case,
- Test Case 2: the method call foo(1, 1, 1, 1, 1.), expected return value of 1. With this method call, we have achieved 100% statement coverage because we have now executed the program statements on lines 6-12.

Branch Coverage

- *Branch coverage is a measure of the percentage of the decision points (Boolean expressions) of the program have been evaluated as both true and false in test cases.*
- *The small program* in Figure 3 has two decision points – one on line 3 and the other on line 7.
  - **Line 3 if (a == 0)** {
  - **Line 7 if ((a==b) OR ((c == d) AND bug(a)))** {
- For decision/branch coverage, we evaluate an entire Boolean expression as one true-or-false predicate even if it contains multiple logical-and or logical-or operators – as in line 7.
- We need to ensure that each of these predicates (compound or single) is tested as both true and false. Table 1 shows our progress so far:

Branch Coverage

Table 1: Decision Coverage

Line #	Predicate	True	False
3	(a==0)	Test Case 1 **foo(0, 0, 0, 0, 0)** **return 0**	Test Case 2 **foo(1, 1, 1, 1, 1)** **return 1**
7	((a==b)OR((c==d)AND bug(a)))	Test Case 2 **foo(1, 1, 1, 1, 1)** **return 1**	

- Therefore, we currently have executed three of the four necessary conditions; we have achieved 75% branch coverage thus far.
- We add Test Case 3 to bring us to 100% branch coverage: foo(1, 2, 1, 2, 1). When we look at the code to calculate an expected return value, we realize that this test case uncovers a previously undetected division-by-zero problem on line 10!
- In many cases, the objective is to achieve 100% branch coverage in your testing, though in large systems only 75%-85% is practical. Only 50% branch coverage is practical in very large systems of 10 million source lines of code or more (Beizer, 1990).

Condition Coverage

- *Condition coverage is a measure of percentage of Boolean sub-expressions of the program that have been evaluated as both true or false outcome [applies to compound predicate] in test cases. Notice that in* line 7 there are three sub-Boolean expressions to the larger statement (a==b), (c==d), and bug(a).
- Condition coverage measures the outcome of each of these sub-expressions independently tested as both true and false. We consider our progress thus far in Table 2.

Table 2: Condition Coverage

Predicate	True	False
(a==b)	Test Case 2 **foo(1, 1, x, x, 1)** **return 0**	Test Case 3 **foo(1, 2, 1, 2, 1)** **Division by zero**
(c==d)		Test Case 3 **foo(1, 2, 1, 2, 1)** **Division by zero**

Condition Coverage

- At this point, our condition coverage is only 50%. The true condition (c==d) has never been tested. Additionally, short-circuit Boolean has prevented the method bug(int) from ever being executed. We examine our available information on the bug method and determine that is should return a value of true when passed a value of a=1.
- Write Test Case 4 to address test (c==d) as true: **foo(1, 2, 1, 1, 1), expected return value 1.**
- However, when we actually run the test case, the function bug(a) actually returns false, which causes our actual return value (division by zero) to not match our expected return value. This allows us to detect an error in the bug method. Without the addition of condition coverage, this error would not have been revealed.
- To finalize our condition coverage, we must force bug(a) to be false. We again examine our bug() information, which informs us that the bug method should return a false value if fed any integer greater than 1. So we create Test Case 5, foo(3, 2, 1, 1, 1), expected return value "division by error". The condition coverage thus far is shown in Table 3

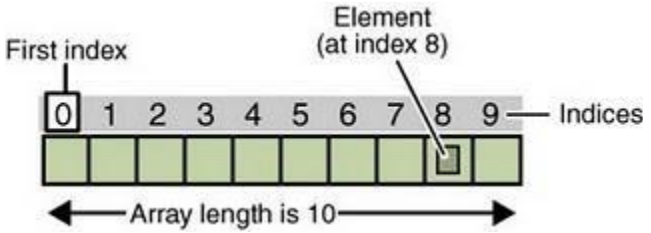
Condition Coverage

Table 3: Condition Coverage

Predicate	True	False
(a==b)	Test Case 2 **foo(1, 1, 1, 1, 1)** **return 0**	Test Case 3 **foo(1, 2, 1, 2, 1)** **Division by zero**
(c==d)	Test Case 4 **foo(1, 2, 1, 1, 1)** **return 1**	Test Case 3 **foo(1, 2, 1, 2, 1)** **Division by zero**
Bug(a)	Test Case 4 **foo(1, 2, 1, 1, 1)** **return 1**	Test Case 5 **foo(3, 2, 1, 1, 1)** **Division by zero**

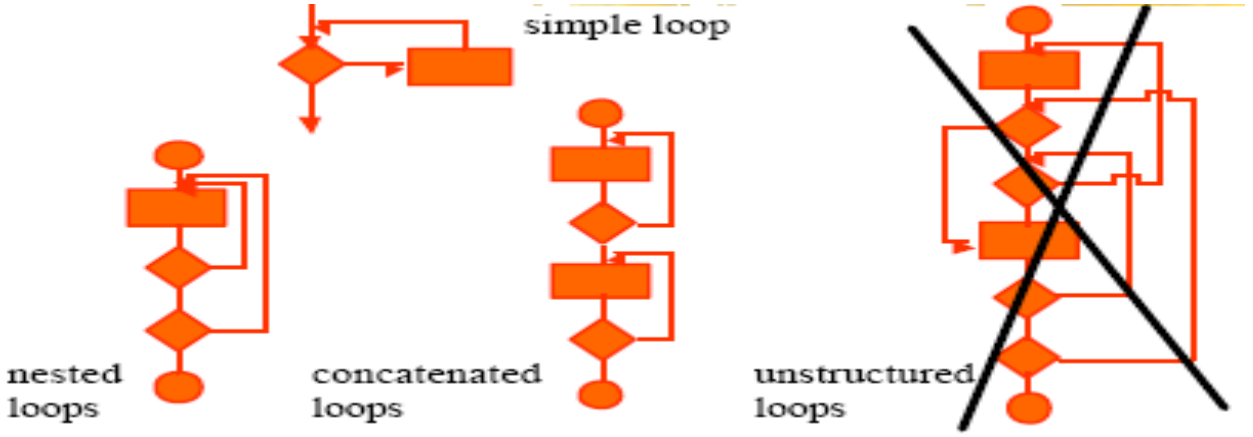
Domain Testing

- Domain testing is an important white box testing method. The goal is to check values taken by a variable, a condition, or an index, and to prove whether they are outside the valid range or not. It also contains checking that the program accepts only valid input , because it is unlikely to get reasonable results if idiocy has been entered. This part can be called "garbage in -- garbage out" testing.
- Domain testing can include the following checks
  - Are all indices used to access an array inside the array's dimensions?
  - Are all those indices integers?



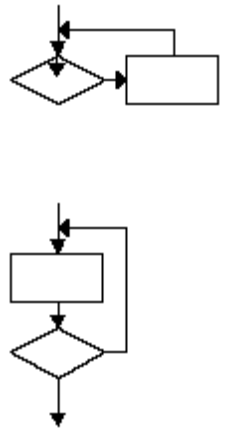
Loop Testing

- Loop testing is a white box testing technique that focuses on the validity of loop constructs.
  - Simple loops
  - Concatenated loops
  - Nested loops



Simple Loops Testing

- The following set of tests can be applied to simple loops, where n is the maximum number of allowable passes through the loop.
- Skip the loop entirely
- Only one pass through the loop
- Two passes through the loop
- m passes through the loop where m<n
- n-1,n,n+1 passes through the loop.



Simple

Nested Loop Testing

- If we were to extend the test approach for simple loops to nested loops the number of possible tests would grow as the level of nesting increases. Beizer suggest an approach that will help to reduce the number of tests:
- Start at the innermost loop. Set all other loops to min. values
- Conduct simple loop tests for the innermost loop while holding the outer loops at their min. iteration parameter values.
- Work outward conducting tests for the next loop, but keeping all other outer loops at min. values and other nested loops to typical values.
- Continue until all loops have been tested.

Data Flow Testing

- Data flow testing is a structural test technique which aims to execute subpaths from points where each variable in a component is defined to points where it is referenced.
- These subpaths are known as definition-use pairs (du-pairs). The different data flow coverage criteria require different du-pairs and subpaths to be executed.
- Data flow testing is a powerful tool to detect improper use of data values due to coding errors.
- Almost every programmer has made this type of mistake:

```
main()
        {
                int x;
                if (x==42)
                { ...}
        }
```

- The mistake is referencing the value of a variable without first assigning a value to it.

Existence Possibilities of a Variable

- Variables that contain data values have a defined life cycle. They are created, they are used, and they are killed (destroyed).
- Three possibilities exist for the first occurrence of a variable through a program path:
- 1.~d: the variable does not exist (indicated by the ~), then it is defined (d)
- 2.~u: the variable does not exist, then it is used (u)
- 3.~k: the variable does not exist, then it is killed or destroyed (k)
- The first is correct. The variable does not exist and then it is defined.
- The second is incorrect. A variable must not be used before it is defined.
- The third is incorrect. Destroying a variable before it is created is indicative of a programming error.

Sequence of Possibilities

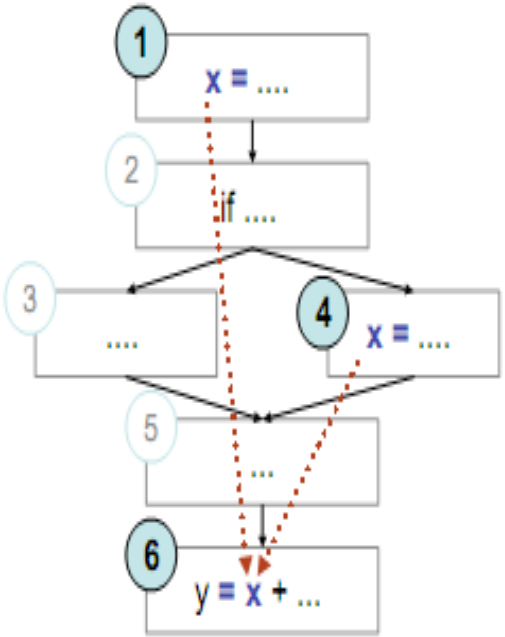
- Now consider the following time-sequenced pairs of defined (d), used (u), and killed (k):
  - **dd:** Defined and defined again-not invalid but suspicious. Probably a programming error.
  - **du:** Defined and used-perfectly correct. The normal case.
  - **dk:** Defined and then killed-not invalid but probably a programming error.
  - **ud:** Used and defined-acceptable.
  - **uu:** Used and used again-acceptable.
  - **uk:** Used and killed-acceptable.
  - **kd:** Killed and defined-acceptable. A variable is killed and then redefined.
  - **ku:** Killed and used-a serious defect. Using a variable that does not exist or is undefined is always an error.
  - **kk:** Killed and killed-probably a programming error.

Terms used in data flow testing

- DU pair: a pair of definition and use for some variable, such that at least one DU path exists from the definition to the use
  - x = … is a definition of x
  - = … x … is a use of x
- DU path: a DU path for a variable is a path from the defining node to the usage node, thus the "flow of data".
- All-du-path (All-Definition Use-Path) coverage testing involves :
  - Identifying all du pairs in the program.
  - Create a path for each du pair.
  - Produce test data for testing the path.

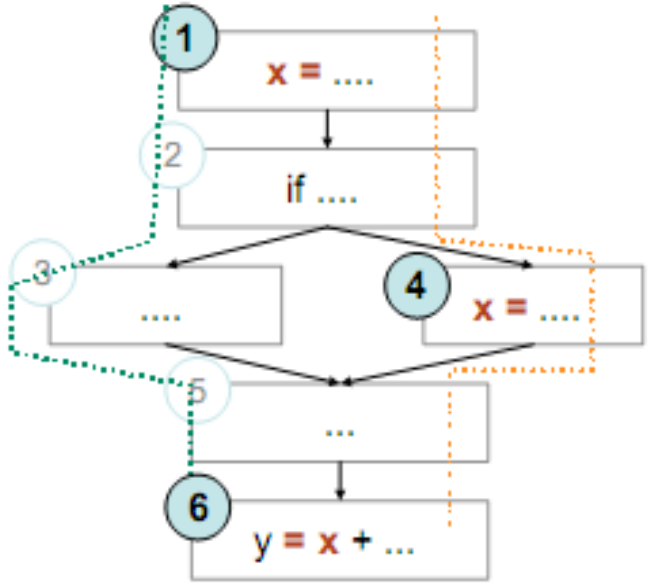
Data Flow Coverage Concept

- Value of x at 6 could be computed at 1 or at 4
- Bad computation at 1 or 4 could be revealed only if they are used at 6
- (1,6) and (4,6) are def-use (DU) pairs
  - defs at 1,4
  - use at 6



Definition Clear Path

- 1,2,3,5,6 is a definition-clear path from 1 to 6
  - x is not re-assigned between 1 and 6
- 1,2,4,5,6 is not a definition-clear path from 1 to 6
  - the value of x is "killed" (reassigned) at node 4



References

[1] Software Engineering by Roger Pressman

[2] http://www.freetutes.com/systemanalysis/sa9-white-box-testing.html

[3] http://docs.ncover.com/best-practices/code-quality-metrics/

[4] http://en.wikipedia.org/wiki/Unit_testing

[5] https://users.cs.jmu.edu/bernstdh/web/common/help/stubs-and-drivers.php

[6] http://www.cs.gmu.edu/~mcjunkin/cs112lectures/Stubs.htm

[7] http://agile.csc.ncsu.edu/SEMaterials/WhiteBox.pdf

[8] http://soft-testing.blogspot.com/2007/12/loop-testing.html