

**Week 1**

**Software testing:** process of executing program/system with intent of finding errors

**Fault:** incorrect portions of code (can be missing as well as incorrect)

**Failure:** observable correct behaviour of program

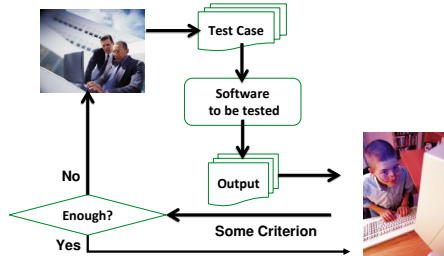
**Error:** cause of fault, something bad programmer did (conceptual, typo, etc)

**Bug:** informal term for fault

**Test case:** set of test inputs, execution conditions, expected results developed for particular objective, such as to exercise particular program path or verify compliance with specific requirement

## A Typical Software Testing Process

### Test case generation



**Testing:** find inputs that cause failure of software, failure unknown, performed by testers

**Debugging:** process of finding & fixing fault given failure, failure is known, performed by devs

**Black-Box/Functional Testing:** identify functions & design test cases to check whether functions are correctly performed by software (formal & informal specs)

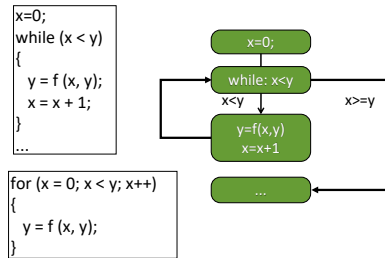
**Equivalence partitioning:** divide into partitions, select 1 test case from each partition, partitions must be disjointed (no input belongs to more than 1 partition) & all partitions must cover entire input domain

**Equivalence partitioning examples:** isEven then even & odd, password min 8 & max 12 characters then less than, valid, more than

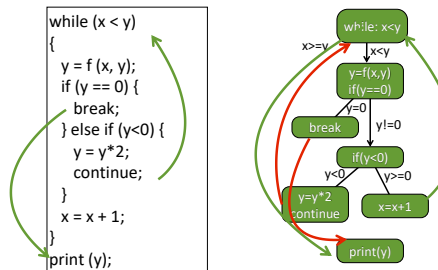
**Boundary-Value analysis:** partition input domain, identify boundaries, select test data (for range  $[R_1, R_2]$  less than  $R_1$ , equal to  $R_1$ , between, equal to  $R_2$ , greater than  $R_2$ , for unordered sets select in & not in, for equality select equal & not equal, for sets, lists select empty & not empty)

**White box/structural testing:** generate test cases based on program structure, abstract program to control flow graph (node is sequence of statements, edge is transfers of control)

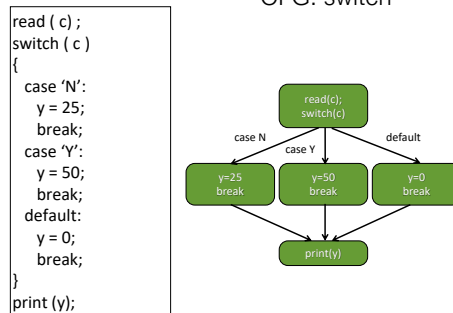
## CFG : while and for loops



## CFG: break and continue



## CFG: switch



**Coverage types:** statement, branch, path (infinite if loop exists), strictly subsumes all beforehand

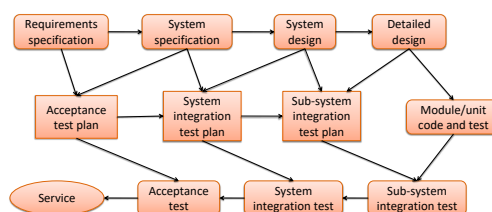
## Week 2

**Test oracle:** expected output of software for given input, part of test case

**Test driver:** software framework that can load collection of test cases or test suite

**Test suite:** collection of test cases

## The V-model of development



## Testing types:

**Unit/Module:** test single module in isolated environment, use drivers & stubs for isolation

**Integration:** test parts of system by combining modules, integrated collection of modules tested as group or partial system

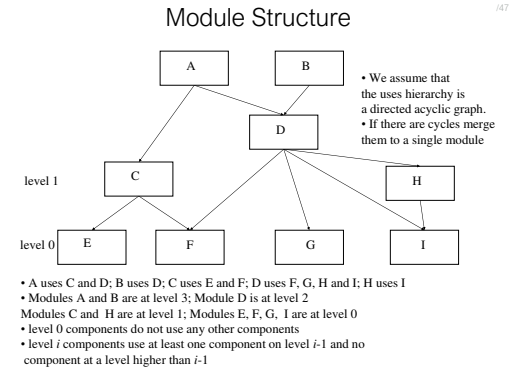
**System:** test system as a whole after integration phase

**Acceptance:** test system as a whole to find out if it satisfies requirements specifications

**Driver:** program that calls interface procedures of module being tested & reports results, simulates module that calls module currently being tested, also provides access to global variables for module under test

**Stub:** program that has same interface as module being used by module being tested but simpler, simulates module called by module being tested

**Mock objects:** create object that mimics behaviour needed for testing



## Integration types:

**Bottom-Up:** only terminal modules tested in isolation, requires drivers but not stubs (since lower levels are tested already)

**Top-down:** modules tested in isolation are modules at highest level, requires stubs but not drivers

**Sandwich:** begin both bottom-up & top-down, meet at predetermined point in middle

**Big bang:** every module unit tested, then integrate all at once, no driver or stub needed but may be hard to isolate bugs

**System/Acceptance testing:** can construct test case based on requirements specifications, main purpose is to assure that system meets requirements, alpha testing performed within development organisation, beta testing performed by select group of friendly customers

## Week 3

**Basis Path Testing:** between branch & path coverage, fulfills branch testing & tests all independent paths that could be used to construct any arbitrary path through computer program

**Independent path:** includes some vertices/edges not covered in other path

**Cyclomatic complexity:**  $e - n + 2p$ ,  $e$  is edges,  $n$  is nodes,  $p$  is number of connected components, or  $1 + d$ ,  $d$  is loops or decision points, upper bound on number of test cases to guarantee coverage of all statements

**Decision Coverage:** executing true/false of decision

**Condition Coverage:** executing true/false of each condition

**Condition/Decision Coverage:** DC & CC, better than either

**Multiple Condition Coverage:** whether every possible combination of boolean sub-expressions occurs, test cases are truth table,  $2^n$  test cases for  $n$  conditions

**Modified C/DC:** for each basic condition  $C$ , 2 test cases, values of all evaluated conditions except  $C$  are the same, compound decision as a while evaluates to true for 1 & false for the other, subsumed by MCC & subsumes CC, DC, C/DC, stronger than statement & branch

**MC/DC coverage:** each entry & exit point invoked, each decision takes every possible outcome, each condition in a decision takes every possible outcome, each condition in decision is shown to independently affect outcome of decision, independence of condition is shown by proving that only one condition changes at a time

## MC/DC: linear complexity

•  $N+1$  test cases for  $N$  basic conditions

(( (a || b) && c) || d) && e

Test Case	a	b	c	d	e	outcome
(1)	true	--	true	--	true	true
(2)	false	true	--	true	true	true
(3)	true	--	false	true	true	true
(4)	true	--	true	--	false	false
(5)	true	--	false	false	--	false
(6)	false	false	--	false	--	false

• Underlined values independently affect the output of the decision

Table 1. Types of Structural Coverage

Coverage Criteria	Statement Coverage	Decision Coverage	Condition Coverage	Condition/Decision Coverage	MC/DC	Multiple Condition Coverage
Every point of entry and exit in the program has been invoked at least once	*	*	*	*	*	*
Every statement in the program has been invoked at least once	*					
Every decision in the program has taken all possible outcomes at least once		*	*	*	*	*
Every condition in a decision in the program has taken all possible outcomes at least once			*	*	*	*
Every condition in a decision has been shown to independently affect that decision's outcome					*	!
Every combination of condition outcomes within a decision has been invoked at least once						*

## Week 4

**Dataflow Coverage:** considers how data gets accessed & modified in system & how it can get corrupted

**Common access-related bugs:** using undeclared

finned/uninitialized variable, deallocating/reinitialising variable before constructed/initialised/used, deleting collection object leaving members inaccessible

**Variable definition:** defined whenever value modified (LHS of assignment, input statement, call-by-reference)

**Variable use:** used whenever value read (RHS of assignment, call-by-value, branch statement predicate)

**p-use:** use in predicate of branch statement

**c-use:** any other use

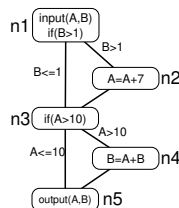
**Use & redefine in single statement:** both sides of assignment, call-by-reference

**du-pair:** with respect to variable  $v$  is a pair  $(d, u)$  such that  $d$  is a node defining  $v$ ,  $u$  is a node/edge using  $v$  (if p-use  $u$  is outgoing edge of predicate), there is a def-clear path with respect to  $v$  from  $d$  to  $u$

**Definition clear:** with respect to variable  $v$  if no variable re-definition of  $v$  on path

## Identifying du-pairs – variable B

du-pair	path(s)
(1,4)	<1,2,3,4>
	<1,3,4>
(1,5)	<1,2,3,5>
	<1,3,5>
(1,<1,2>)	<1,2>
(1,<1,3>)	<1,3>
(4,5)	<4,5>

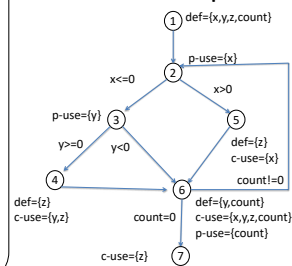


```

1 begin
2  float x,y,z=0.0;
3  int count;
4  input(x,y,count);
5  do{
6    if(x<0) {
7      if(y<0) {
8        z=y*z+1;
9      }
10   }
11  else{
12    z=1/x;
13  }
14  y=x*y+z;
15  count=count-1;}
16  while(count>0)
17  output(z);
18 end

```

## Another Example



### Dataflow test coverage criteria:

**All-Defs:** for every variable  $v$ , at least one def-clear path from every definition of  $v$  to at least one c-use or p-use of  $v$  must be covered

**All-P/C-Uses:** for every variable  $v$ , at least one def-clear path from every definition of  $v$  to every p/c-use of  $v$  must be covered

**All-Uses:** all du-pairs covered

**Notations:**

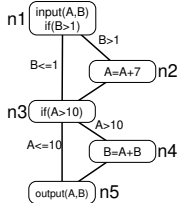
$d_1(x)$ : definition of variable  $x$  in node  $i$

$u_i(x)$ : use of variable  $x$  in node  $i$   $dcu(d_i(x)) = dcu(x, i)$ : set of c-uses with respect to  $d_1(x)$

$dpu(d_i(x)) = dpu(x, i)$ : set of p-uses with respect to  $d_1(x)$

## Identifying du-pairs – variable A

du-pair	path(s)
(1,2)	<1,2>
(1,4)	<1,3,4>
(1,5)	<1,3,4,5>
	<1,3,5>
(1,<3,4>)	<1,3,4>
(1,<3,5>)	<1,3,5>
(2,4)	<2,3,4>
(2,5)	<2,3,4,5>
	<2,3,5>
(2,<3,4>)	<2,3,4>
(2,<3,5>)	<2,3,5>



**All-C-Uses for above:**  $dcu(x,1) + dcu(y,1) + dcu(y,6) + dcu(z,1) + dcu(z,4) + dcu(z,5) + dcu(count,1) + dcu(count,6) = 2 + 2 + 2 + 2 + 3 + 3 + 3 + 1 + 1 = 17$

**All-P-Uses for above:**  $dpu(x,1) + dpu(y,1) + dpu(y,6) + dpu(z,1) + dpu(z,4) + dpu(z,5) + dpu(count,1) + dpu(count,6) = 2 + 2 + 2 + 0 + 0 + 0 + 2 + 2 = 10$  (note this includes using the initial count definition even though it will always be redefined (-1) before the comparison)

## Relationships among some of the coverage criteria

