Scheme of Evaluation

Internal Assessment Test II – April 2019

Sub:	Software Testing							**Code:**	15IS63
Date:	16/04/2019	Duration:	90mins	Max Marks:	50	**Sem:**	VI	**Branch:**	ISE

Note: Answer Any Five Questions

Question #	**Description**	**Marks Distribution**		**Max Marks**
1	**What is a decision table? Design a decision table for next date problem and derive the test cases.** • Defining decision table with example • Equivalence classes for NextDate • Decision Table • Test Cases	2M 2M 3M 3M	10 M	10 M
2	**Analyze the commission problem from the perspective of data flow testing, derive the DU paths for the variables locks, stocks, barrels, sales and commission and check whether they are definition clear paths** • Program code • Program graph • DU and DC paths for the variables	3 M 2 M 5 M	10 M	10 M
3	**Derive Basis paths for triangle problem using Mc Cabe's method and write the test cases for the derived paths.** • Program code DD-Path graph • Cyclomatic Complexity • Identification of paths • Test Cases	5M 1M 3M 1M	10 M	10 M
4	**Explain with example i) Block Coverage ii) Condition Coverage** i) Block Coverage definition with equation Example ii) Condition Coverage definition with equation	2.5M 2.5M 2.5M	10 M	10 M

		Example	2.5 M		
5		**Write a short note on i) Scaffolding ii) Test Oracles.** • Scaffolding explanation with types • Test Oracles explanation with types	5M 5 M	10 M	10 M
6		**Explain slice based testing with example** • Definition of slice • Example code • Identifying slices • Lattices	2M 3M 3M 2M	10 M	10 M
7		**List out the test coverage metrics and explain metric based testing.** • Listing of Test coverage metrics • Explanation of metric based testing	3M 7M	5 M	10 M

IAT-2 Solution
Software Testing (15IS63)
April 2018-19

1) What is a decision table? Design a decision table for next date problem and derive the test cases.

A decision table has four portions: the part to the left of the bold vertical line is the stub portion; to the right is the entry portion. The part above the bold horizontal line is the condition portion, and below is the action portion. Thus, we can refer to the condition stub, the condition entries, the action stub, and the action entries. A column in the entry portion is a rule. Rules indicate which actions, if any, are taken for the circumstances indicated in the condition portion of the rule.

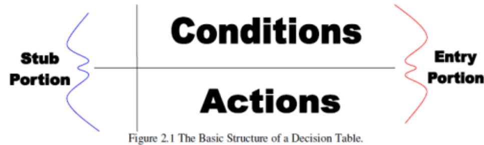


Figure 2.1 The Basic Structure of a Decision Table.

M1 = {month: month has 30 days}
M2 = {month: month has 31 days except December}
M3 = {month: month is December}
M4 = {month: month is February}
D1 = {day: $1 \le$ day ≤ 27}
D2 = {day: day = 28}
D3 = {day: day = 29}
D4 = {day: day = 30}
D5 = {day: day = 31}
Y1 = {year: year is a leap year}

Y2 = {year: year is a common year}

Table 7.14 Decision Table for NextDate Function

	1	2	3	4	5	6	7	8	9	10		
c1: Month in	M1	M1	M1	M1	M1	M2	M2	M2	M2	M2		
c2: Day in	D1	D2	D3	D4	D5	D1	D2	D3	D4	D5		
c3: Year in	—	—	—	—	—	—	—	—	—	—		
Actions												
a1: Impossible					X							
a2: Increment day	X	X	X			X	X	X	X			
a3: Reset day				X						X		
a4: Increment month				X						X		
a5: Reset month												
a6: Increment year												
	11	12	13	14	15	16	17	18	19	20	21	22
c1: Month in	M3	M3	M3	M3	M3	M4	M4	M4	M4	M4	M4	M4
c2: Day in	D1	D2	D3	D4	D5	D1	D2	D2	D3	D3	D4	D5
c3: Year in	—	—	—	—	—	—	Y1	Y2	Y1	Y2	—	—
Actions												
a1: Impossible										X	X	X
a2: Increment day	X	X	X	X		X	X					
a3: Reset day					X			X	X			
a4: Increment month								X	X			
a5: Reset month					X							
a6: Increment year					X							

Table 7.16 Decision Table Test Cases for NextDate

Case ID	Month	Day	Year	Expected Output
1–3	4	15	2001	4/16/2001
4	4	30	2001	5/1/2001
5	4	31	2001	Invalid input date
6–9	1	15	2001	1/16/2001
10	1	31	2001	2/1/2001
11–14	12	15	2001	12/16/2001
15	12	31	2001	1/1/2002
16	2	15	2001	2/16/2001
17	2	28	2004	2/29/2004
18	2	28	2001	3/1/2001
19	2	29	2004	3/1/2004
20	2	29	2001	Invalid input date
21, 22	2	30	2001	Invalid input date

2) Analyze the commission problem from the perspective of data flow testing, derive the DU paths for the variables locks, stocks, barrels, sales and commission and check whether they are definition clear paths.

Variable name	Defined at node	Used at Node
lprice	7	24
sprice	8	25
bprice	9	26
tlocks	10,16	16,21,24
tstocks	11,17	17,22,25
tbarrels	12,18	18,23,26
locks	13,19	14,16
stocks	15	17
barrels	15	18
lsales	24	27
ssales	25	27
bsales	26	27
sales	27	28,29,33,34,37,39
comm	31,32,33,36,37,39	32,33,37,42

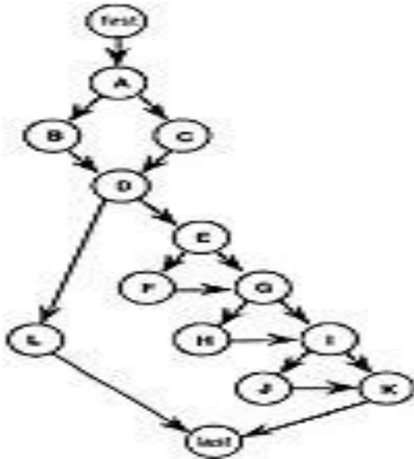
Test case id	Description	Variables Path (Beginning, End nodes)	Du Paths

1	Check for lock price variable DEF(lprice,7) and USE(lprice,24)	(7 , 24)	<7-8-9-10-11-12-13-14-15-16-17-18-19-20-21-22-23-24>
2	Check for Stock price variable DEF(sprice,8) and USE(sprice,25)	(8 , 25)	<8-9-10-11-12-13-14-15-16-17-18-19-20-21-22-23-24-25>
3	Check for barrel price variable DEF(bprice,9) and USE(bprice,26)	(9 , 26)	<9-10-11-12-13-14-15-16-17-18-19-20-21-22-23-24-25-26>
4	Check for total locks variable DEF((tlocks,10) and DEF(tlocks,16)) and 3 usage node(USE(tlocks,16),USE(tlocks,21),USE(tlocks,24)	(10 , 16)	<10-11-12-13-14-15-16>
		(10 , 21)	<10-11-12-13-14-15-16-17-18-19-20-14-21>
		(10 , 24)	<10-11-12-13-14-15-16-17-18-19-20-14-21-22-23-24>
		(16 , 16)	<16-16>
		(16 , 21)	<16-17-18-19-14-21>
5	Check for total stocks variable DEF((tstocks,11) and DEF(tstocks,17)) and 3 usage node(USE(tstocks,17),USE(tstocks,22),USE(tstocks,25)	(11 , 17)	<11-12-13-14-15-16-17>
		(11 , 22)	<11-12-13-14-15-16-17-18-19-20-21-14-21>
		(11, 25)	<11-12-13-14-15-16-17-18-19-20-21-14-21-23-24-25>
		(17 , 17)	<17-17>
		(17 , 22)	<17-18-19-20-14-21-22>

3) Derive Basis paths for triangle problem using Mc Cabe's method and write the test cases for the derived paths.

```
1)  program triangle (input, output) ;
2)  VAR a, b, c      : integer;
3) IsATriangle : boolean;
4)  BEGIN
5)  writeln('Enter three integers which are sides of a triangle:');
6)  readln (a,b,c);
7)  writeln('Side A is ',a, 'Side B is ',b, 'side C is ',c);
8)  IF (a < b + c) AND (b < a + c) AND (c < a + b)
9) THEN IsATriangle :=TRUE
10)ELSE IsATriangle := FALSE ;
11) IF IsATriangle
12)THEN
13)BEGIN
14)IF (a = b) XOR (a = c) XOR (b = c) AND NOT((a=b) AND (a=c))
15) THEN Writeln ('Triangle is Isosceles') ;
16)IF (a = b) AND (b = c)
```

17) THEN Writeln ('Triangle is Equilateral') ;
18) IF (a <> b) AND (a <> c) AND (b <> c)
19) THEN Writeln ('Triangle is Scalene') ;
20) END
21) ELSE WRITELN('Not a Triangle') ;
22) END.



p1: A-B-D-E-G-I-J-K-Last
p2: A-C-D-E-G-I-J-K-Last
p3: A-B-D-L-Last
p4: A-B-D-E-F-G-I-J-K-Last
p5: A-B-D-E-F-G-H-I-J-K-Last
p6: A-B-D-E-F-G-H-I-K-Last

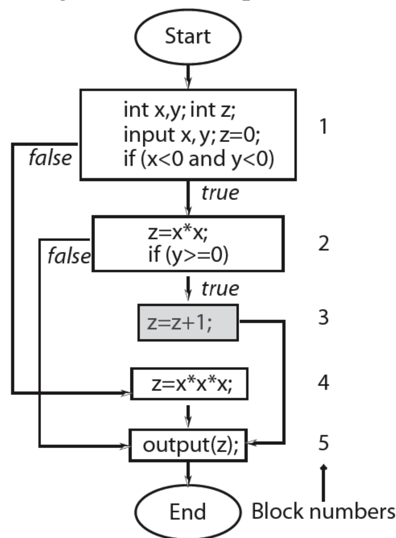
If you follow paths p2, p3, p4, p5, and p6, you find that they are all infeasible. Path p2 is infeasible, because passing through node C means the sides are not a triangle, so none of the sequel decisions can be taken. Similarly, in p3, passing through node B means the sides do form a triangle, so node L cannot be traversed. The others are all infeasible because they involve cases where a triangle is of two types (e.g., isosceles and equilateral). The problem here is that there are several inherent dependencies in the triangle problem. One is that if three integers constitute sides of a triangle, they must be one of the three possibilities: equilateral, isosceles, or scalene. A second dependency is that the three possibilities are mutually exclusive: if one is true, the other two must be false.

fp1: A-C-D-L-Last	(Not a triangle)
fp2: A-B-D-E-F-G-I-K-Last	(Isosceles)
fp3: A-B-D-E-G-H-I-K-Last	(Equilateral)

fp4: A-B-D-E-G-I-J-K-Last	(Scalene)

4) Explain with example i) Block Coverage ii) Condition Coverage

The **block coverage** of T with respect to (P, R) is computed as **Bc/(Be -Bi)** , where Bc is the number of blocks covered, Bi is the number of unreachable blocks, and Be is the total number of blocks in the program, i.e. the size of the block coverage domain.
T is considered adequate with respect to the block coverage criterion if the statement coverage of T with respect to (P, R) is 1.



Coverage domain: Be={1, 2, 3, 4, 5}

$$T_2 = \left\{ \begin{array}{lll} t_1: & < x = -1 & y = -1 > \\ t_2: & < x = -3 & y = -1 > \\ t_3: & < x = -1 & y = -3 > \end{array} \right\}$$

Blocks covered:
t1: Blocks 1, 2, 5
t2, t3: same coverage as of t1.
Be=5 , Bc=3, Bi=1.
Block coverage for T2= 3/(5-1)=0.75.
Hence T2 is not adequate for (P, R) with respect to the block coverage criterion.

The **condition coverage** of T with respect to (P, R) is computed as Cc/(Ce -Ci) , where Cc is the number of simple conditions covered, Ci is the number of infeasible simple conditions, and |Ce is the total number of simple conditions in the program, i.e. the size of the condition coverage domain.
T is considered adequate with respect to the condition coverage criterion if the condition coverage of T with respect to (P, R) is 1.

```
1    begin
2      int x, y, z;
3      input (x, y);
4      if(x<0 and y<0)
5        z=foo1(x,y);
6      else
7        z=foo2(x,y);
8      output(z);
9    end
```

Consider the test set:
$T = \{t_1 :< x = -3, y = -2 > \ t_2 :< x = -4, y = -2 >\}$

Check that T is adequate with respect to the statement, block, and decision coverage criteria and the program behaves correctly against t1 and t2.
Cc=1, Ce=2, Ci=0. Hence condition coverage for T=0.5.

5) Write a short note on i) Scaffolding ii) Test Oracles.

Scaffolding:
Code produced to support development activities (especially testing)

- Not part of the "product" as seen by the end user
- May be temporary (like scaffolding in construction of buildings

Includes

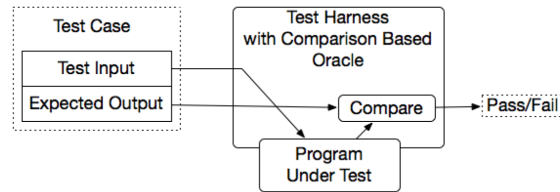
- Test harnesses
    i. Substitutes for other parts of the deployed environment
        Ex: Software simulation of a hardware device
-
- Drivers:
    i. A "main" program for running a test
        1. May be produced before a "real" main program
        2. Provides more control than the "real" main program
            a. To driver program under test through test cases
- Stubs
    i. Substitute for called functions/methods/objects

Generic or Specific scaffolding:

- How general should scaffolding be?
    - We could build a driver and stubs for each test case
    - ... or at least factor out some common code of the driver and test management (e.g., JUnit)
    - ... or further factor out some common support code, to drive a large number of test cases from data (as in DDSteps)
    - ... or further, generate the data automatically from a more abstract model (e.g., network traffic model)
- A question of costs and re-use
    - Just as for other kinds of software

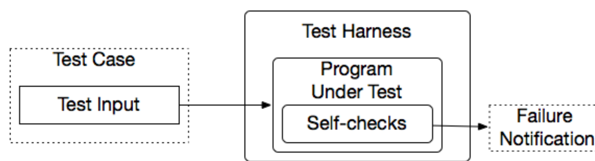
Test Oracles: If a software test is a sequence of activities (*stimuli and observations*), an **oracle** is a predicate that determines whether a given sequence is acceptable or not

Comparison-based oracle



- With a comparison-based oracle, we need predicted output for each input
  - Oracle compares actual to predicted output, and reports failure if they differ
- Fine for a small number of hand-generated test cases
  - E.g., for hand-written JUnit test cases

Self-Checking Code as Oracle



- An oracle can also be written as *self-checks*
  - Often possible to judge correctness without predicting results
- Advantages and limits: Usable with large, automatically generated test suites, but often only a *partial* check
  - e.g., structural invariants of data structures
  - recognize *many* or *most* failures, but not all

6) Explain slice based testing with example.

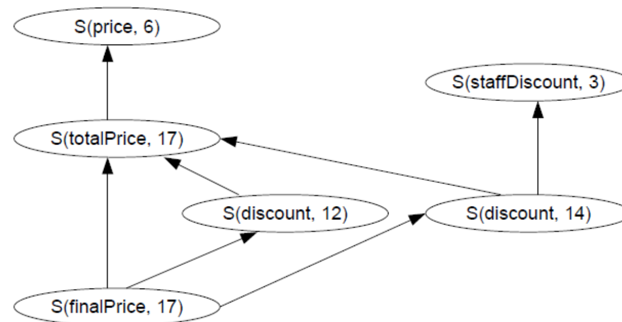
Program slice: Given a program *P* and a set *V* of variables in *P*, a *slice on the variable set V at statement n*, written *S*(*V*, *n*), is the set of all statement fragments in *P* that contribute to the values of variables in *V* at node *n*

```
1  program Example()
2  var staffDiscount, totalPrice, finalPrice, discount, price
3  staffDiscount = 0.1
4  totalPrice = 0
5  input(price)
6  while(price != -1) do
7      totalPrice = totalPrice + price
8      input(price)
9  od
10 print("Total price: " + totalPrice)
11 if(totalPrice > 15.00) then
12     discount = (staffDiscount * totalPrice) + 0.50
13 else
14     discount = staffDiscount * totalPrice
15 fi
16 print("Discount: " + discount)
17 finalPrice = totalPrice - discount
```

So, for example, with respect to the price variable given in the example in section 2, the following are slices for each use of the variable:
• S(price, 5) = {5}
• S(price, 6) = {5, 6, 8, 9}
• S(price, 7) = {5, 6, 8, 9}
• S(price, 8) = {8}
The program slice, as already mentioned, allows the programmer to focus specifically on the code that is relevant to a particular variable at a certain point. However, the program slice concept also allows the programmer to generate a lattice of slices: that is, a graph showing the subset relationship between the different slices. For instance, looking at the previous example for the variable price, the slices S(price, 5) and S(price, 8) are subsets of S(price, 7).

With respect to a program as a whole, certain variables may be related to the values of other variables: for instance, a variable that contains a value that is to be returned at the end of the execution may use the values of other variables in the program. For instance, in the main example in this document, the finalPrice variable uses the totalPrice variable, which itself uses the price variable. The finalPrice variable also uses the discount variable, which uses the staffDiscount and totalPrice variables – and so on.
Therefore, the slices of the totalPrice and discount variables are a subset of the slice of the finalPrice variable at lines 17 and 18, as they both contribute to the value. This subset relationship 'ripples down' to the other variables, according to the use-relationship described.
This is shown visually in the following example:

- S(staffDiscount, 3) = {3}
- S(totalPrice, 4) = {4}
- S(totalPrice, 7) = {4, 5, 6, 7, 8}
- S(totalPrice, 11) = {4, 5, 6, 7, 8}
- S(discount, 12) = {3, 4, 5, 6, 7, 8, 11, 12}
- S(discount, 14) = {3, 4, 5, 6, 7, 8, 13, 14}
- S(finalPrice, 17) = {3, 4, 5, 6, 7, 8, 11, 12, 13, 14, 17}



Therefore, the lattice of slices for the finalPrice variable is as shown in Figure above. This relationship, as shown in the lattice diagram, can feasibly help during testing, particularly if there's a fault. For instance, if there is an error in the slice of finalPrice, then, by testing the different subset slices, you can eliminate them from the possible sources of the error (for instance, the error may be generated from an incorrect calculation of the discount, for instance).

7) List out the test coverage metrics and explain metric based testing

C_0: Every statement

C_1 : Every DD-Path (predicate outcome)

C_{1p}: Every predicate to each outcome

C_2: C_1 coverage + loop coverage

C_d: C_1 coverage + every dependent pair of DD-Paths

C_{MCC}: Multiple condition coverage

C_{ik}: Every program path that contains up to k repetitions of a loop (usually k = 2)

C_{stat}: Statistically significant" fraction of paths

C_∞ : All possible execution paths

Miller's test coverage metrics are based on program graphs in which nodes are full statements, whereas our formulation allows statement fragments to be nodes. For the remainder of this section, the statement fragment formulation is "in effect".

Statement and Predicate Testing

Statement fragments can be considered to be single nodes. In our triangle example is a complete Pascal IF-THEN-ELSE statement. If we required nodes to correspond to full statements, we could execute just one of the decision alternatives and satisfy the statement coverage criterion. Because we allow statement fragments, it is "natural" to divide such a statement into three nodes. Doing so results in predicate outcome coverage. Whether or not our convention is followed, these coverage metrics require that we find a set of test cases such that, when executed, every node of the program graph is traversed at least once.

DD-Path Testing

When every DD-path is traversed (the $C1$ metric), we know that each predicate outcome has been executed; this amounts to traversing every edge in the DD-path graph (or program graph). Therefore, the $C1$ metric is exactly our *G*chain metric. For if–then and if–then–else statements, this means that both the true and the false branches are covered ($C1$p coverage). For CASE statements, each clause is covered. Beyond this, it is useful to ask how we might test a DD-path. Longer DD-paths generally represent complex computations, which we can rightly consider as individual functions. For such DD-paths, it may be appropriate to apply a number of functional tests, especially those for boundary and special values.

Dependent Pairs of DD-Paths

The C_d metric foreshadows the dataflow testing. The most common dependency among pairs of DD-Paths is the define/reference relationship, in which a variable is defined (receives a value) in one DD-Path and is referenced in another DD-Path. The importance of these dependencies is that they are closely related to the problem of

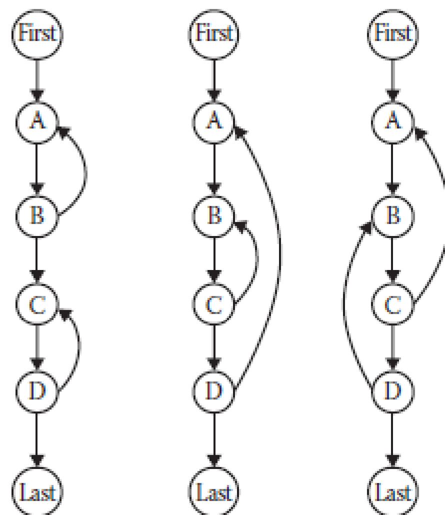
infeasible paths. We have good examples of dependent pairs of DD-Paths: in Figure 9.4, B and D are such a pair, so are DD-Paths C and L. Simple DD- Path coverage might not exercise these dependencies, thus a deeper class of faults would not be revealed.

Multiple Condition Coverage

Look closely at the compound conditions in DD-Paths A and E. Rather than simply traversing such predicates to their TRUE and FALSE outcomes, we should investigate the different ways that each outcome can occur. One possibility is to make a truth table; a compound condition of three simple conditions would have eight rows, yielding eight test cases. Another possibility is to reprogram compound predicates into nested simple IF-THEN-ELSE logic, which will result in more DD-Paths to cover. We see an interesting trade-off: statement complexity versus path complexity. Multiple condition coverage assures that this complexity isn't swept under the DD-Path coverage rug.

Loop Coverage

The condensation graphs provide us with an elegant resolution to the problems of testing loops. Loop testing has been studied extensively, and with good reason — loops are a highly fault prone portion of source code. To start, an amusing taxonomy of loops occurs (Beizer, 1984): concatenated, nested, and horrible, shown in Figure



Concatenated loops are simply a sequence of disjoint loops, while nested loops are such that one is contained inside another. Knotted (Beizer calls them "horrible") loops cannot occur when the structured programming precepts are followed, but they can occur in languages like Java with try/catch. When it is possible to branch into (or out from) the middle of a loop, and these branches are internal to other loops, the result is Beizer's knotted loop. We can also take a modified boundary value approach, where the loop index is given its minimum,

nominal, and maximum values. We can push this further to full boundary value testing and even robustness testing. If the body of a simple loop is a DD-path that performs a complex calculation, this should also be tested, as discussed previously. Once a loop has been tested, the tester condenses it into a single node. If loops are nested, this process is repeated starting with the innermost loop and working outward. This results in the same multiplicity of test cases we found with boundary value analysis, which makes sense, because each loop index variable acts like an input variable. If loops are knotted, it will be necessary to carefully analyze them in terms of the data flow method