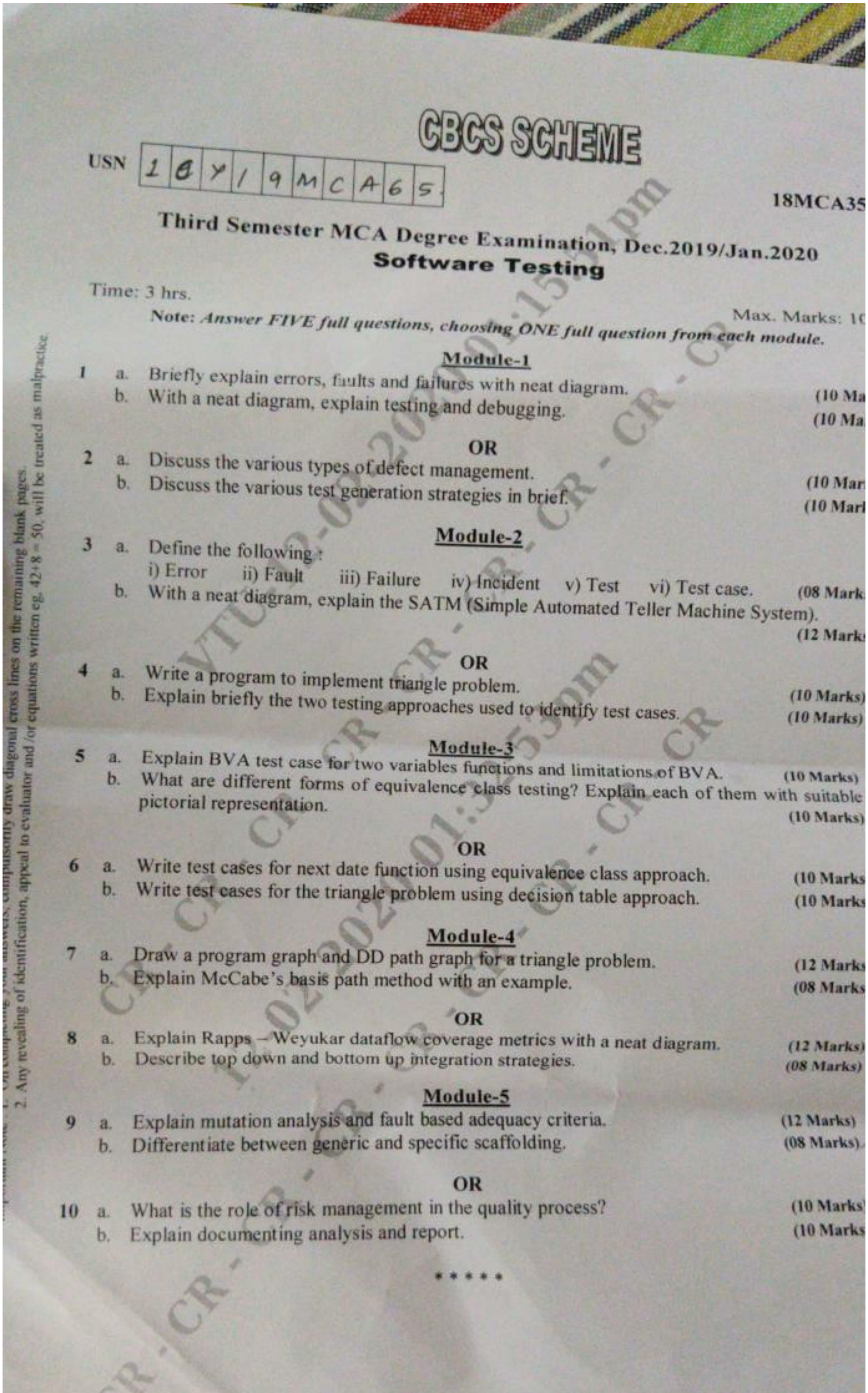


CBCS SCHEME

USN 1 G Y 1 9 M C A 6 5

18MCA35

Third Semester MCA Degree Examination, Dec.2019/Jan.2020
Software Testing

Time: 3 hrs. Max. Marks: 1(

Note: *Answer FIVE full questions, choosing ONE full question from each module.*

Module-1

1 a. Briefly explain errors, faults and failures with neat diagram. (10 Ma

b. With a neat diagram, explain testing and debugging. (10 Ma

OR

2 a. Discuss the various types of defect management. (10 Mar

b. Discuss the various test generation strategies in brief. (10 Marl

Module-2

3 a. Define the following :
i) Error ii) Fault iii) Failure iv) Incident v) Test vi) Test case. (08 Mark

b. With a neat diagram, explain the SATM (Simple Automated Teller Machine System). (12 Mark

OR

4 a. Write a program to implement triangle problem. (10 Marks)

b. Explain briefly the two testing approaches used to identify test cases. (10 Marks)

Module-3

5 a. Explain BVA test case for two variables functions and limitations of BVA. (10 Marks)

b. What are different forms of equivalence class testing? Explain each of them with suitable pictorial representation. (10 Marks)

OR

6 a. Write test cases for next date function using equivalence class approach. (10 Marks

b. Write test cases for the triangle problem using decision table approach. (10 Marks

Module-4

7 a. Draw a program graph and DD path graph for a triangle problem. (12 Marks

b. Explain McCabe's basis path method with an example. (08 Marks

OR

8 a. Explain Rapps – Weyukar dataflow coverage metrics with a neat diagram. (12 Marks)

b. Describe top down and bottom up integration strategies. (08 Marks)

Module-5

9 a. Explain mutation analysis and fault based adequacy criteria. (12 Marks)

b. Differentiate between generic and specific scaffolding. (08 Marks)

OR

10 a. What is the role of risk management in the quality process? (10 Marks)

b. Explain documenting analysis and report. (10 Marks

* * * * *

1. On completing your answers, compulsorily draw diagonal cross lines on the remaining blank pages.
2. Any revealing of identification, appeal to evaluator and /or equations written eg. 42+8 = 50, will be treated as malpractice.

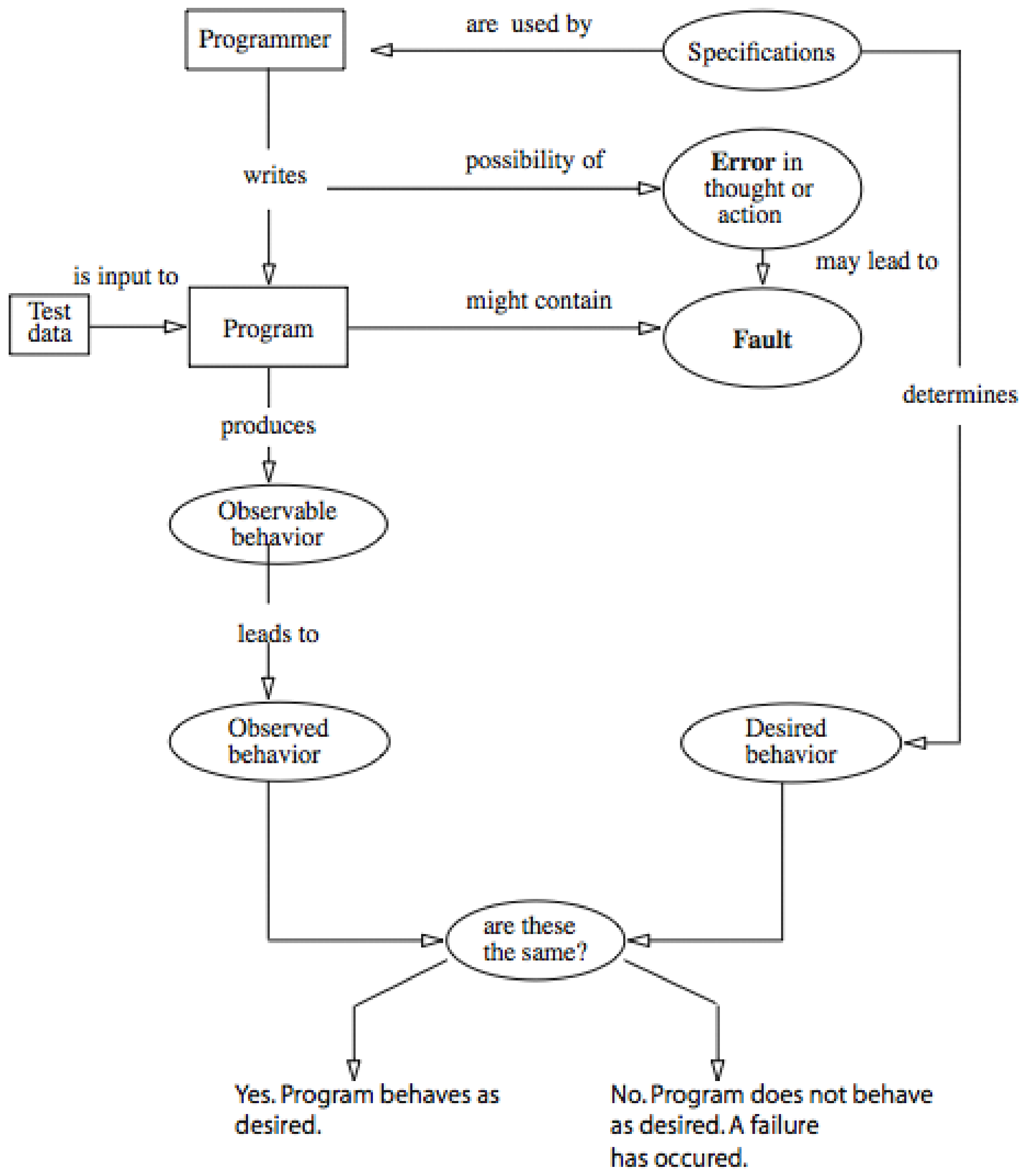
Q1a:

Ans:

Error—People make errors. A good synonym is *mistake*. When people make mistakes while coding, we call these mistakes *bugs*. Errors tend to propagate; a requirements error may be magnified during design and amplified still more during coding.

Fault—A fault is the result of an error. It is more precise to say that a fault is the representation of an error, where representation is the mode of expression, such as narrative text, Unified Modeling Language diagrams, hierarchy charts, and source code. *Defect* is a good synonym for fault, as is *bug*. Faults can be elusive. An error of omission results in a fault in which something is missing that should be present in the representation. This suggests a useful refinement; we might speak of faults of commission and faults of omission. A fault of commission occurs when we enter something into a representation that is incorrect. Faults of omission occur when we fail to enter correct information. Of these two types, faults of omission are more difficult to detect and resolve.

Failure—A failure occurs when the code corresponding to a fault executes. Two subtleties arise here: one is that failures only occur in an executable representation, which is usually taken to be source code, or more precisely, loaded object code; the second subtlety is that this definition relates failures only to faults of commission. How can we deal with failures that correspond to faults of omission? We can push this still further: what about faults that never happen to execute, or perhaps do not execute for a long time? Reviews prevent many failures by finding faults; in fact, well-done reviews can find faults of omission.

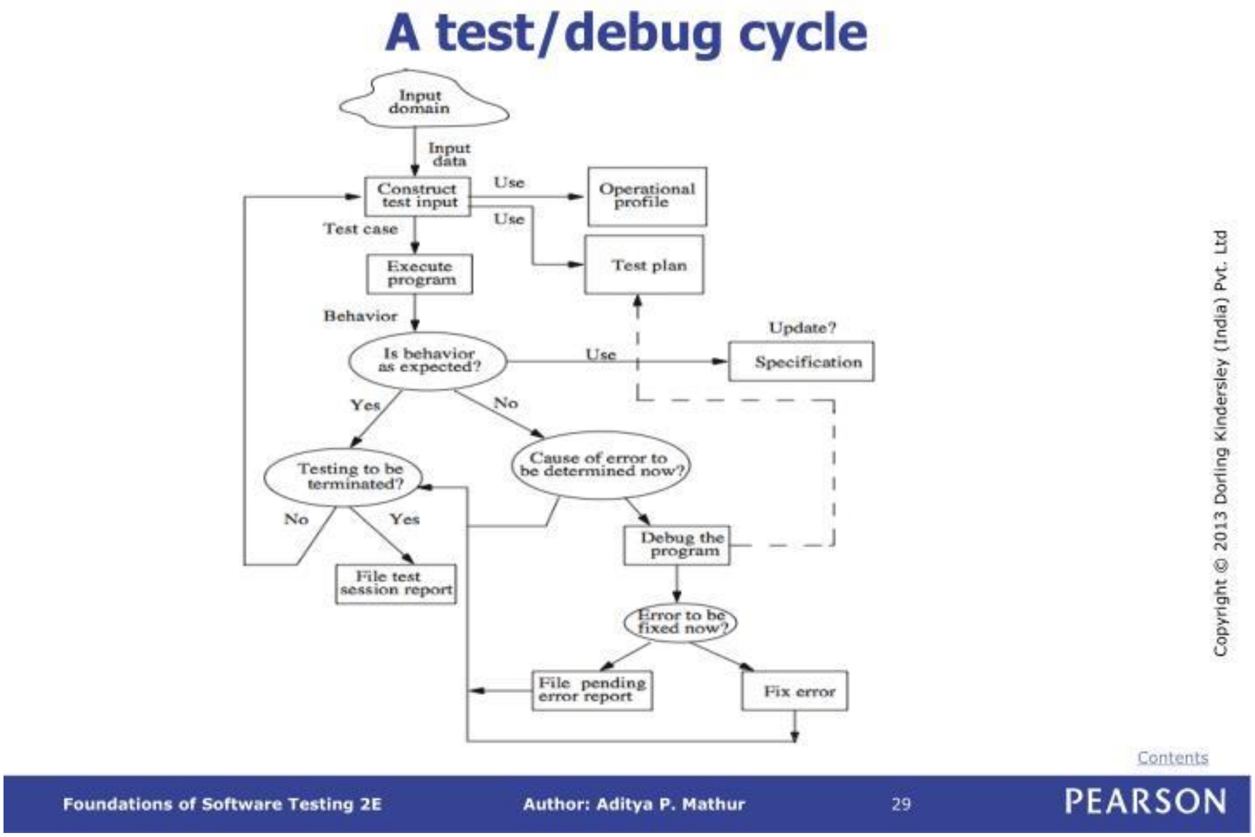


1b:

Ans:

Testing is the process of determining if a program has any errors.

When testing reveals an error, the process used to determine the cause of this error and to remove it, is known as **debugging**.



A test/debug cycle

Contents

Foundations of Software Testing 2E Author: Aditya P. Mathur 29 PEARSON

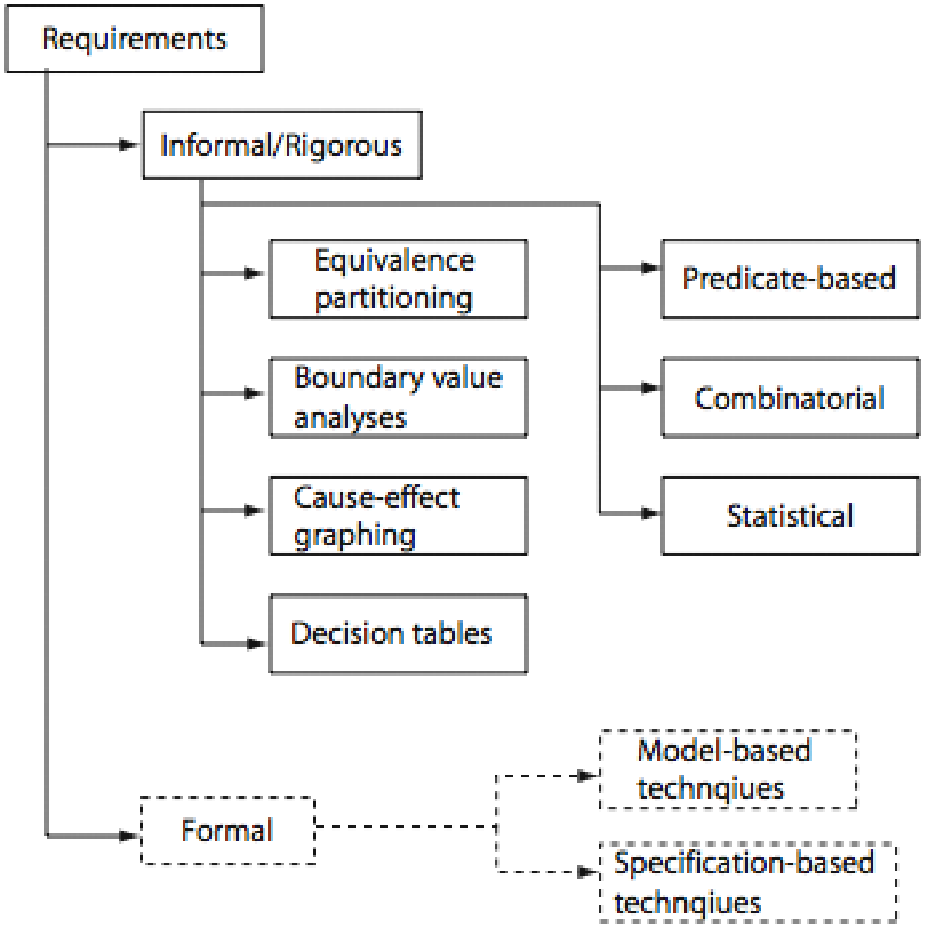
2a:

Ans:

2 b:

Ans: Test generation techniques described belong to the black-box testing category.

These techniques are useful during functional testing where the objective is to test whether or not an application, unit, system, or subsystem, correctly implements the functionality as per the given requirements



Equivalence partitioning: Test selection using equivalence partitioning allows a tester to subdivide the input domain into a relatively small number of sub-domains, say N>1, as shown (next slide (a)).

Errors at the boundaries: Experience indicates that programmers make mistakes in processing values at and near the boundaries of equivalence classes.

Predicates Testing : BOR and BRO for generating tests that are guaranteed to detect certain faults in the coding of conditions. The conditions from which tests are generated might arise from requirements or might be embedded in the program to be tested.

Q 3 a:

Ans:

Error—People make errors. A good synonym is *mistake*. When people make mistakes while coding, we call these mistakes *bugs*. Errors tend to propagate; a requirements error may be magnified during design and amplified still more during coding.

Fault—A fault is the result of an error. It is more precise to say that a fault is the representation of an error, where representation is the mode of expression, such as narrative text, Unified Modeling Language diagrams, hierarchy charts, and source code. *Defect* is a good synonym for fault, as is *bug*. Faults can be elusive. An error of omission results in a fault in which something is missing that should be present in the representation. This suggests a useful refinement; we might speak of faults of commission and faults of omission. A fault of commission occurs when we enter something into a representation that is incorrect. Faults of omission occur when we fail to enter correct information. Of these two types, faults of omission are more difficult to detect and resolve.

Failure—A failure occurs when the code corresponding to a fault executes. Two subtleties arise here: one is that failures only occur in an executable representation, which is usually taken to be source code, or more precisely, loaded object code; the second subtlety is that this definition relates failures only to faults of commission. How can we deal with failures that correspond to faults of omission? We can push this still further: what about faults that never happen to execute, or perhaps do not execute for a long time? Reviews prevent many failures by finding faults; in fact, well-done reviews can find faults of omission.

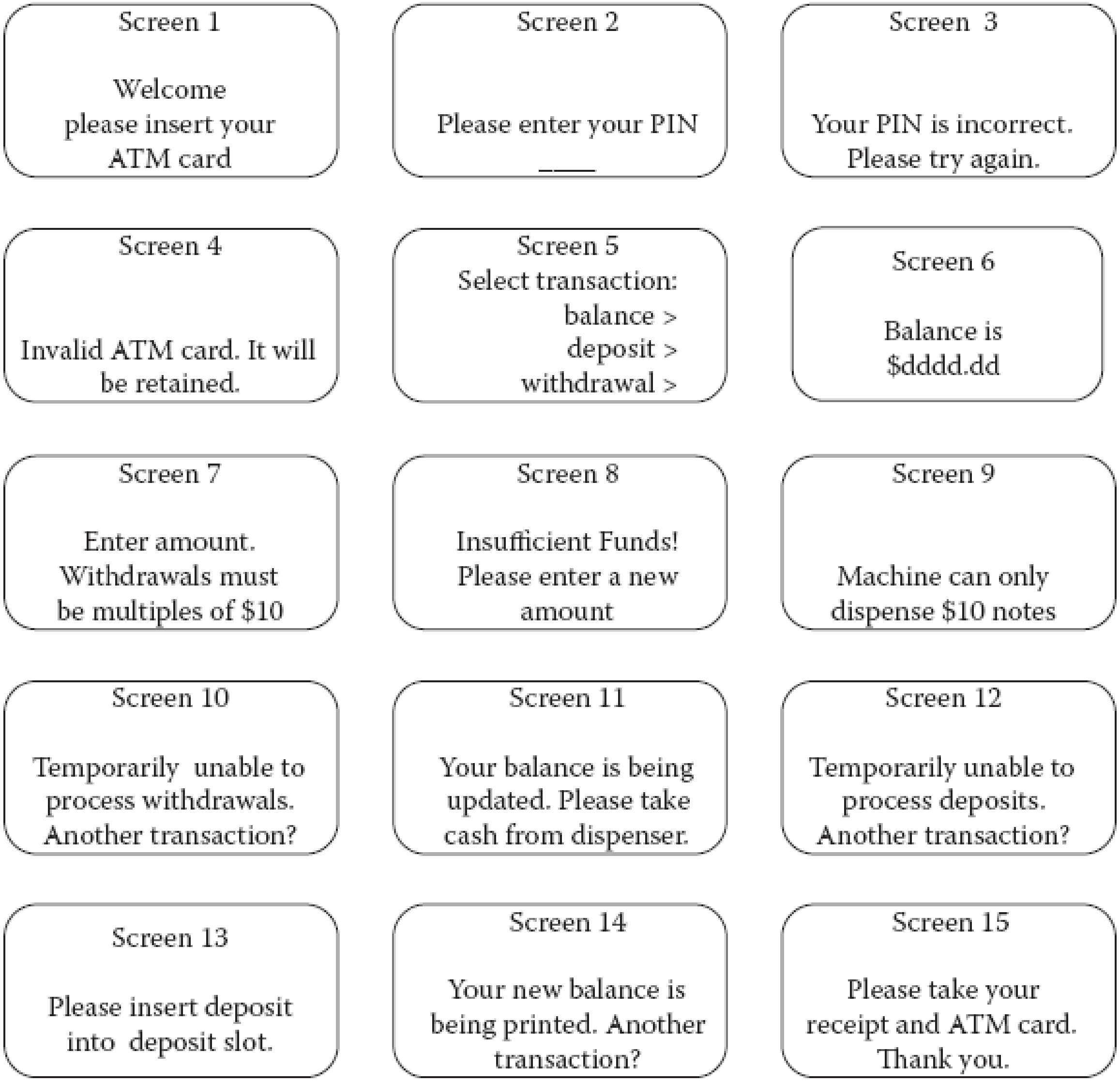
Incident—When a failure occurs, it may or may not be readily apparent to the user (or customer or tester). An incident is the symptom associated with a failure that alerts the user to the occurrence of a failure.

Test is the process of determining if a program has any errors.

Test Case:A test case is a pair of input data and the corresponding program output.Test data are a set of values: one for each input variable.A test set sometimes referred as test suite is a collection of test cases.Test data is an alternate term for test set.Program requirements and test plan help in construction of test data.

3 b:

Ans:



Screen 1

Welcome
please insert your
ATM card

Screen 2

Please enter your PIN
____

Screen 3

Your PIN is incorrect.
Please try again.

Screen 4

Invalid ATM card. It will
be retained.

Screen 5
Select transaction:
balance >
deposit >
withdrawal >

Screen 6

Balance is
$dddd.dd

Screen 7

Enter amount.
Withdrawals must
be multiples of $10

Screen 8

Insufficient Funds!
Please enter a new
amount

Screen 9

Machine can only
dispense $10 notes

Screen 10

Temporarily unable to
process withdrawals.
Another transaction?

Screen 11

Your balance is being
updated. Please take
cash from dispenser.

Screen 12

Temporarily unable to
process deposits.
Another transaction?

Screen 13

Please insert deposit
into deposit slot.

Screen 14

Your new balance is
being printed. Another
transaction?

Screen 15

Please take your
receipt and ATM card.
Thank you.

The SATM system communicates with bank customers via the 15 screens. Using a terminal with features as shown in Figure 2.3, SATM customers can select any of three transaction types: deposits, withdrawals, and balance inquiries. For simplicity, these transactions can only be done on a checking account.

When a bank customer arrives at an SATM station, screen 1 is displayed. The bank customer accesses the SATM system with a plastic card encoded with a personal account number (PAN), which is a key to an internal customer account file, containing, among other things, the customer's name and account information. If the customer's PAN matches the information in the customer account file, the system presents screen 2 to the customer. If the customer's PAN is not found, screen 4 is displayed, and the card is kept.

At screen 2, the customer is prompted to enter his or her personal identification number (PIN).

If the PIN is correct (i.e., matches the information in the customer account file), the system displays screen 5; otherwise, screen 3 is displayed. The customer has three chances to get the PIN correct; after three failures, screen 4 is displayed, and the card is kept. On entry to screen 5, the customer selects the desired transaction from the options shown on screen. If balance is requested, screen 14 is then displayed. If a deposit is requested, the status of the deposit envelope slot is determined from a field in the terminal control file. If no problem is known, the system displays screen 7 to get the transaction amount.

Q4a:
Ans:

```
#include<stdio.h>

int main()

{

    int a,b,c;

    char istriangle;

    printf("enter 3 integers which are sides of triangle\n");
    scanf("%d%d%d",&a,&b,&c);
    printf("a=%d\t,b=%d\t,c=%d",a,b,c);

    // to check is it a triangle or not

    if( a<b+c && b<a+c && c<a+b )

        istriangle='y';

    else

        istriangle ='n';

    ;

    if (istriangle=='y')

        if ((a==b) && (b==c))

            printf("equilateral triangle\n");

        else if ((a!=b) && (a!=c) && (b!=c))

            printf("scalene triangle\n");

          else

            printf("isosceles triangle\n");

    else
```

```
                    printf("Not a triangle\n");

          return 0;

}
```

4 b:
Ans:

BVA test case for two variables functions

In the general application of Boundary Value Analysis can be done in a uniform manner.

The basic form of implementation is to maintain all but one of the variables at their nominal (normal or average) values and allowing the remaining variable to take on its extreme values. The values used to test the extremities are:

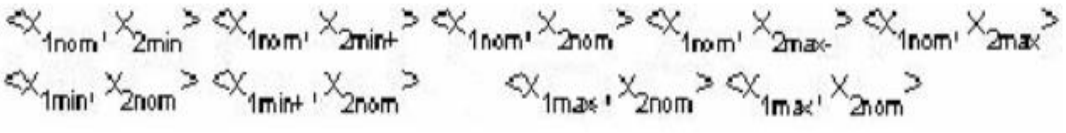
•Min ------------------------------------ - Minimal

•Min+ ------------------------------------ - Just above Minimal

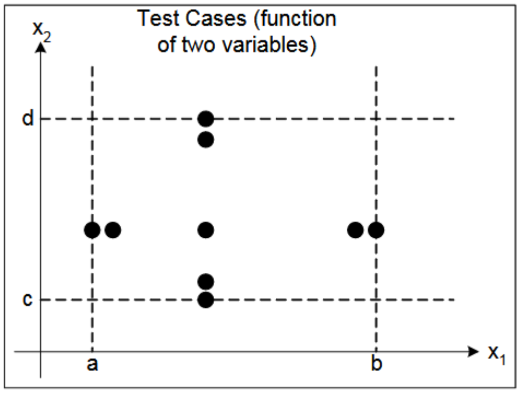
•Nom ------------------------------------ - Average

•Max- ------------------------------------ - Just below Maximum

•Max ------------------------------------ - Maximum



$<x_{1nom}, x_{2min}>$ $<x_{1nom}, x_{2min+}>$ $<x_{1nom}, x_{2nom}>$ $<x_{1nom}, x_{2max-}>$ $<x_{1nom}, x_{2max}>$
$<x_{1min}, x_{2nom}>$ $<x_{1min+}, x_{2nom}>$ $<x_{1max}, x_{2nom}>$ $<x_{1max}, x_{2nom}>$



Limitations of BVA

Boundary Value Analysis works well when the Program Under Test (PUT) is a “function of several independent variables that represent bounded physical quantities” [1]. When these conditions are met BVA works well but when they are not we can find deficiencies in the results. For example the NextDate problem, where Boundary Value Analysis would place an even testing regime equally over the range, tester’s intuition

and common sense shows that we require more emphasis towards the end of February or on leap years.

The reason for this poor performance is that BVA cannot compensate or take into consideration the nature of a function or the dependencies between its variables.

Equivalence Class Test

EC Testing is when you have a number of test items (e.g. values) that you want to test but because of cost (time/money) you do not have time to test them all. Therefore you group the test item into class where all items in each class are suppose to behave exactly the same. The theory is that you only need to test one of each item to make sure the system works.

Example 1

Children under 2 ride the buss for free. Young people pay $10, Adults $15 and Senior Citizen pay $5.
Classes:
Price:0 -> Age:0-1
Price:10 -> Age:2-14
Price:15 -> Age:15-64
Price:5 -> Age:65-infinity

Example 2 (more than one parameter)

Cellphones K80, J64 and J54 run Java 5. K90 and J99 run Java 6. But there are two possible browsers FireFox and Opera, J models run FF and K models run O.
Classes:
Browser:FF, Java:5 -> Phones:J64,J54
Browser:FF, Java:6 -> Phones:J99
Browser:O, Java:5 -> Phones:K80
Browser:O, Java:6 -> Phones:K90

Q5a:

BVA test case for two variables functions

In the general application of Boundary Value Analysis can be done in a uniform manner.

The basic form of implementation is to maintain all but one of the variables at their

nominal (normal or average) values and allowing the remaining variable to take on its

extreme values. The values used to test the extremities are:

•Min ------------------------------------ - Minimal

•Min+ ------------------------------------ - Just above Minimal

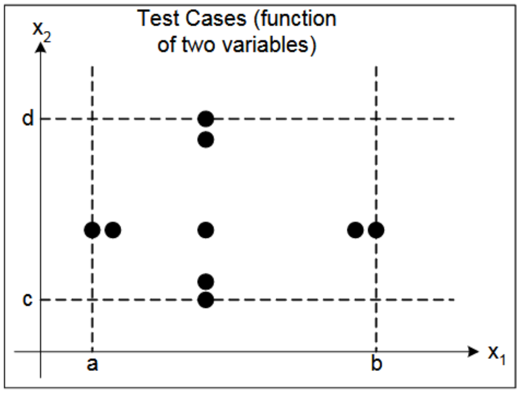
•Nom ------------------------------------ - Average

•Max- ------------------------------------ - Just below Maximum

•Max ------------------------------------ - Maximum

$<x_{1nom}, x_{2min}> <x_{1nom}, x_{2min+}> <x_{1nom}, x_{2nom}> <x_{1nom}, x_{2max-}> <x_{1nom}, x_{2max}>$

$<x_{1min}, x_{2nom}> <x_{1min+}, x_{2nom}> \quad <x_{1max}, x_{2nom}> <x_{1max}, x_{2nom}>$



Limitations of BVA

Boundary Value Analysis works well when the Program Under Test (PUT) is a “function of several independent variables that represent bounded physical quantities” [1]. When these conditions are met BVA works well but when they are not we can find deficiencies in the results. For example the NextDate problem, where Boundary Value Analysis would place an even testing regime equally over the range, tester’s intuition

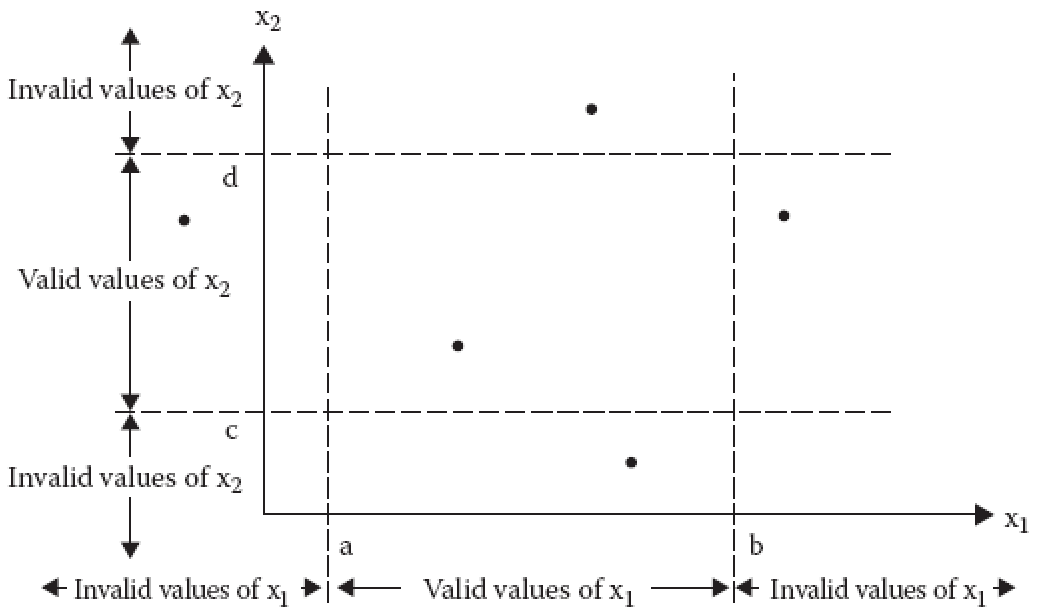
and common sense shows that we require more emphasis towards the end of February or on leap years.

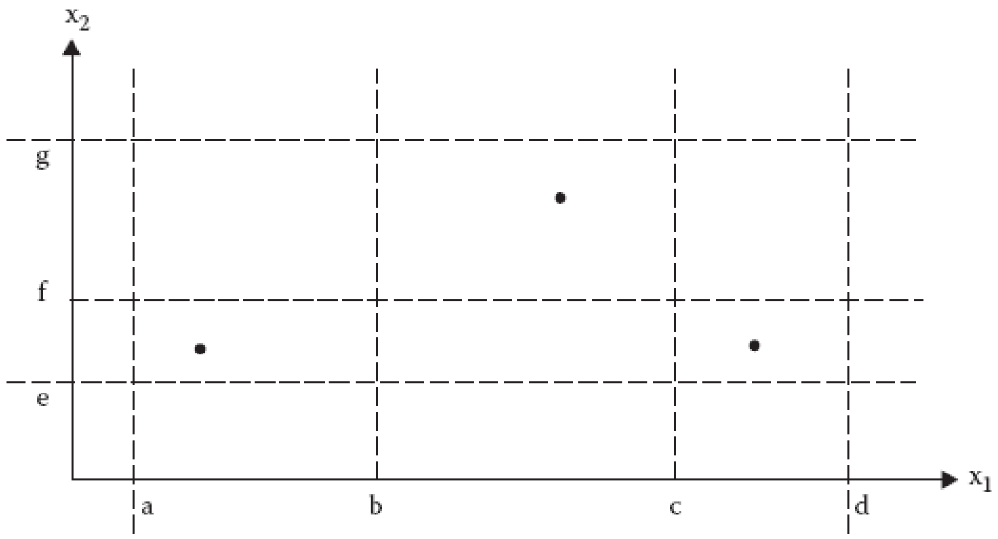
The reason for this poor performance is that BVA cannot compensate or take into consideration the nature of a function or the dependencies between its variables.

5 b:

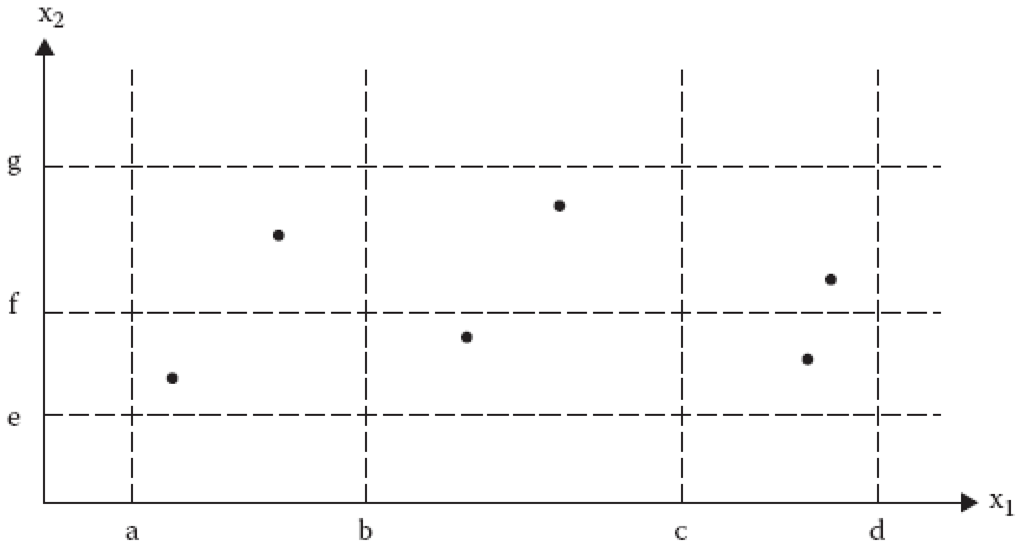
Equivalence Class Test

EC Testing is when you have a number of test items (e.g. values) that you want to test but because of cost (time/money) you do not have time to test them all. Therefore you group the test item into class where all items in each class are suppose to behave exactly the same. The theory is that you only need to test one of each item to make sure the system works.
Example 1
Children under 2 ride the buss for free. Young people pay $10, Adults $15 and Senior Citizen pay $5.
Classes:
Price:0 -> Age:0-1
Price:10 -> Age:2-14
Price:15 -> Age:15-64
Price:5 -> Age:65-infinity

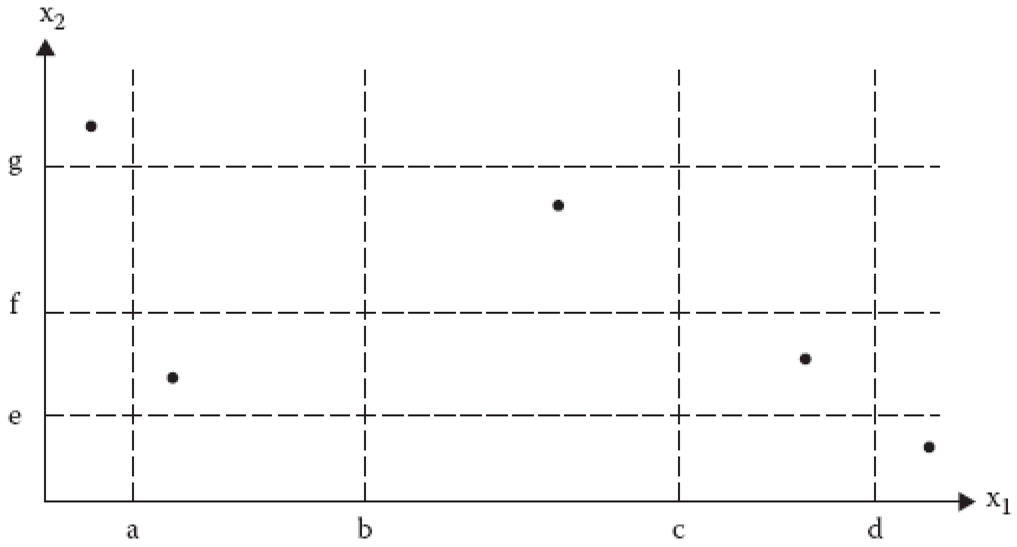




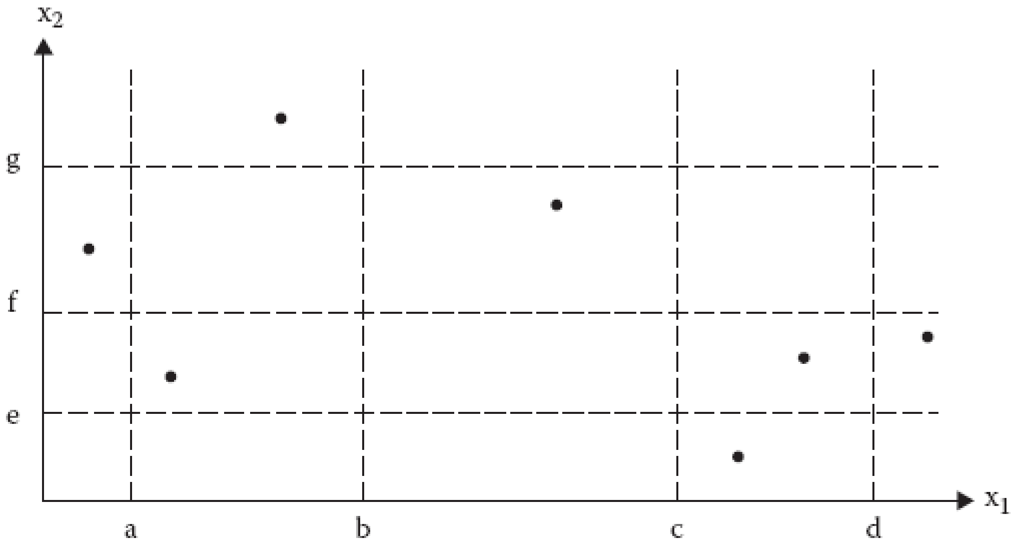
6.2 Weak normal equivalence class test cases.



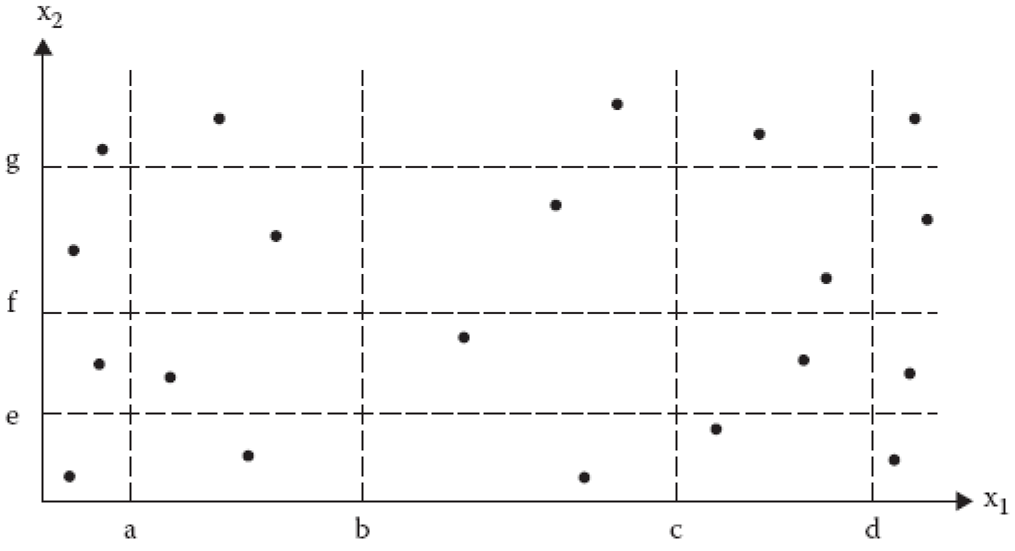
Strong normal equivalence class test cases.



Weak robust equivalence class test cases.



Revised weak robust equivalence class test cases.



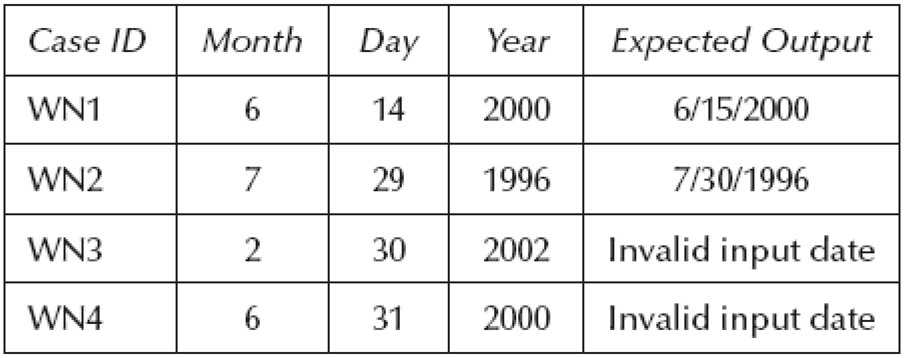
5 Strong robust equivalence class test cases.

Q6 a:
Ans:
we might postulate the following equivalence classes:
M1 = {month: month has 30 days}
M2 = {month: month has 31 days}

M3 = {month: month is February}
D1 = {day: 1 ≤ day ≤ 28}
D2 = {day: day = 29}
D3 = {day: day = 30}
D4 = {day: day = 31}
Y1 = {year: year = 2000}
Y2 = {year: year is a non-century leap year}
Y3 = {year: year is a common year}

By choosing separate classes for 30- and 31-day months, we simplify the question of the last day of the month. By taking February as a separate class, we can give more attention to leap year questions. We also give special attention to day values: days in D1 are (nearly) always incremented, while days in D4 only have meaning for months in M2. Finally, we have three classes of years, the special case of the year 2000, leap years, and non-leap years. This is not a perfect set of equivalence classes, but its use will reveal many potential errors.

These classes yield the following weak normal equivalence class test cases

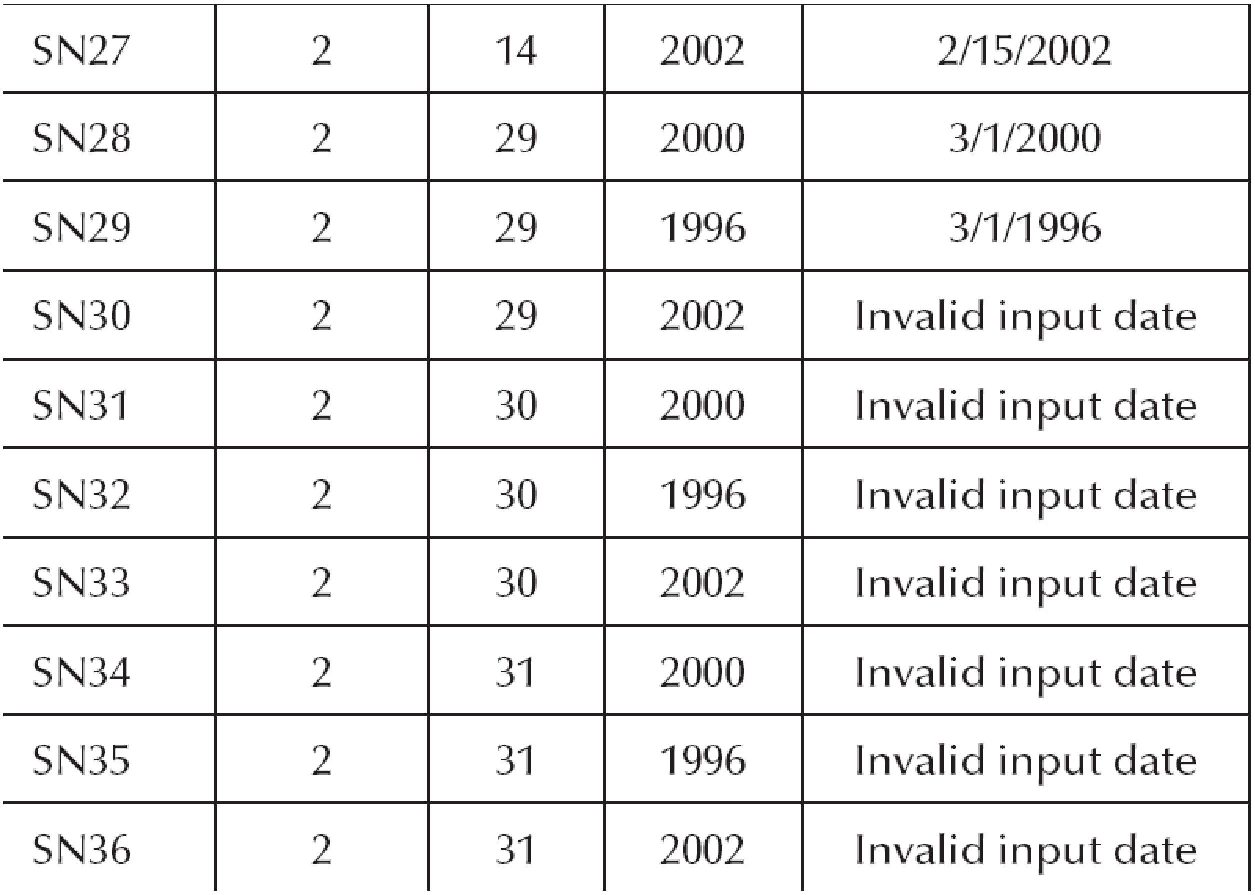


Case ID	*Month*	*Day*	*Year*	*Expected Output*
WN1	6	14	2000	6/15/2000
WN2	7	29	1996	7/30/1996
WN3	2	30	2002	Invalid input date
WN4	6	31	2000	Invalid input date

strong normal equivalence class test cases for the revised classes are as follows:

Case ID	*Month*	*Day*	*Year*	*Expected Output*
SN1	6	14	2000	6/15/2000
SN2	6	14	1996	6/15/1996
SN3	6	14	2002	6/15/2002
SN4	6	29	2000	6/30/2000
SN5	6	29	1996	6/30/1996
SN6	6	29	2002	6/30/2002
SN7	6	30	2000	Invalid input date
SN8	6	30	1996	Invalid input date
SN9	6	30	2002	Invalid input date
SN10	6	31	2000	Invalid input date
SN11	6	31	1996	Invalid input date
SN12	6	31	2002	Invalid input date
SN13	7	14	2000	7/15/2000
SN14	7	14	1996	7/15/1996

Case ID	*Month*	*Day*	*Year*	*Expected Output*
SN15	7	14	2002	7/15/2002
SN16	7	29	2000	7/30/2000
SN17	7	29	1996	7/30/1996
SN18	7	29	2002	7/30/2002
SN19	7	30	2000	7/31/2000
SN20	7	30	1996	7/31/1996
SN21	7	30	2002	7/31/2002
SN22	7	31	2000	8/1/2000
SN23	7	31	1996	8/1/1996
SN24	7	31	2002	8/1/2002
SN25	2	14	2000	2/15/2000
SN26	2	14	1996	2/15/1996



SN27	2	14	2002	2/15/2002
SN28	2	29	2000	3/1/2000
SN29	2	29	1996	3/1/1996
SN30	2	29	2002	Invalid input date
SN31	2	30	2000	Invalid input date
SN32	2	30	1996	Invalid input date
SN33	2	30	2002	Invalid input date
SN34	2	31	2000	Invalid input date
SN35	2	31	1996	Invalid input date
SN36	2	31	2002	Invalid input date

6 b:
We can use a following set of equivalence classes.
Ans: 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 ≤ 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}

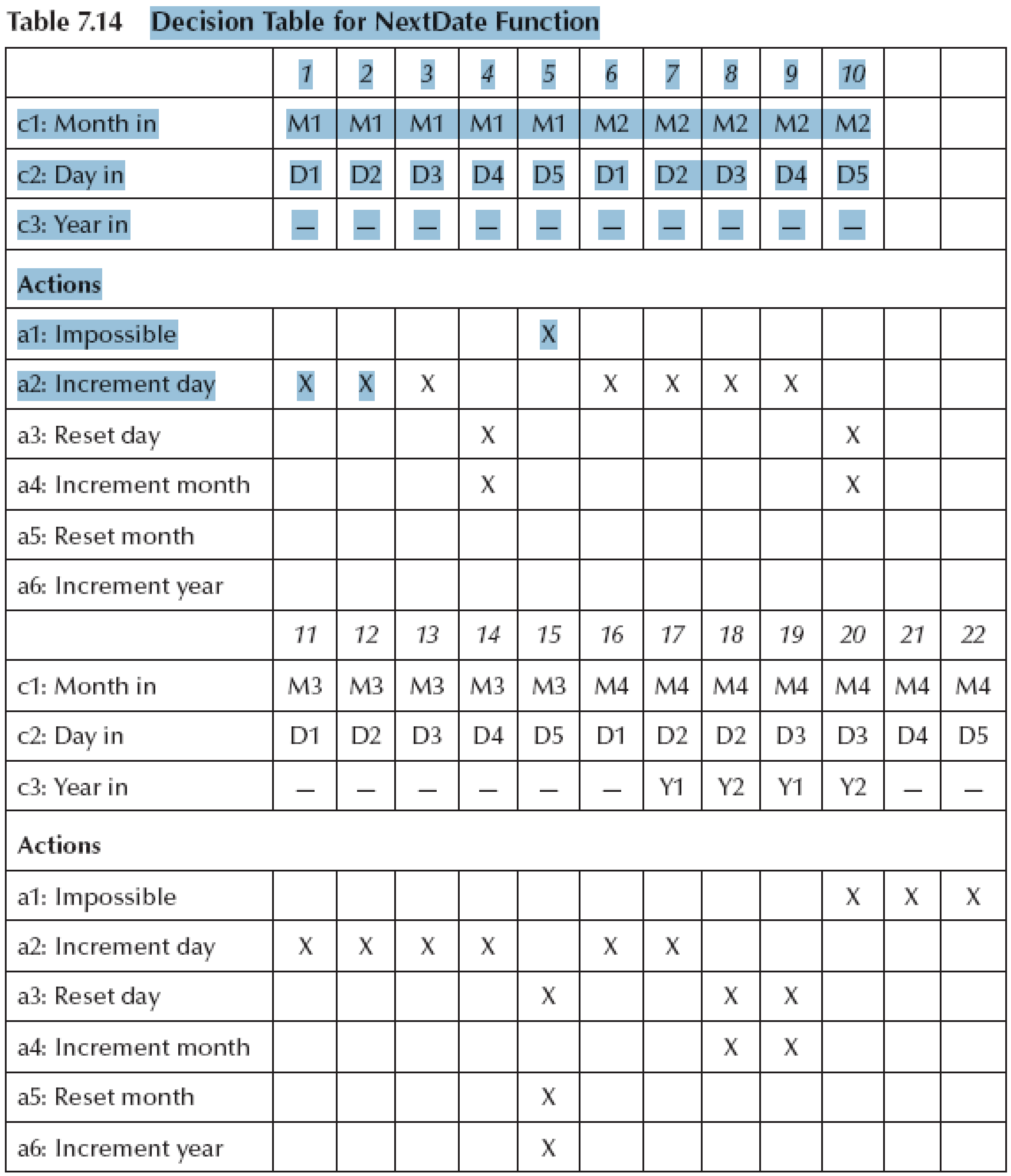


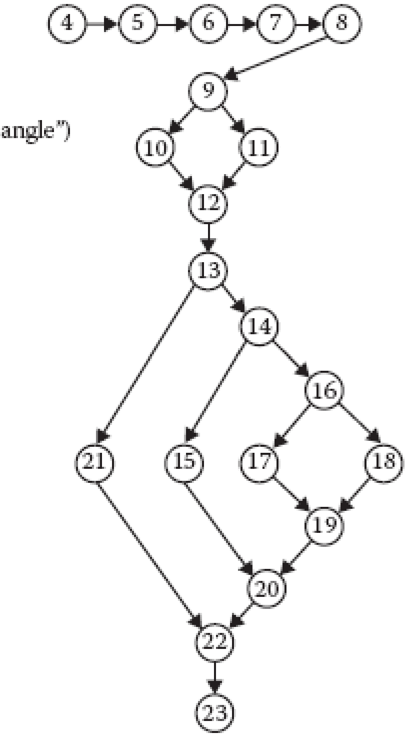
Table 7.14 Decision Table for NextDate Function

	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							

Q7a:

Ans:

```
1  Program triangle2
2  Dim a,b,c As Integer
3  Dim IsATrinagle As Boolean
4  Output("Enter 3 integers which are sides of a triangle")
5  Input(a,b,c)
6  Output("Side A is", a)
7  Output("Side B is", b)
8  Output("Side C is", c)
9  If (a < b + c) AND (b < a + c) AND (c < a + b)
10   Then IsATriangle = True
11   Else IsATriangle = False
12  EndIf
13  If IsATriangle
14   Then  If (a = b) AND (b = c)
15          Then Output ("Equilateral")
16          Else  If (a≠b) AND (a≠c) AND (b≠c)
17                  Then Output ("Scalene")
18                  Else Output ("Isosceles")
19               EndIf
20          EndIf
21   Else  Output("Nota a Triangle")
22  EndIf
23  End triangle2
```



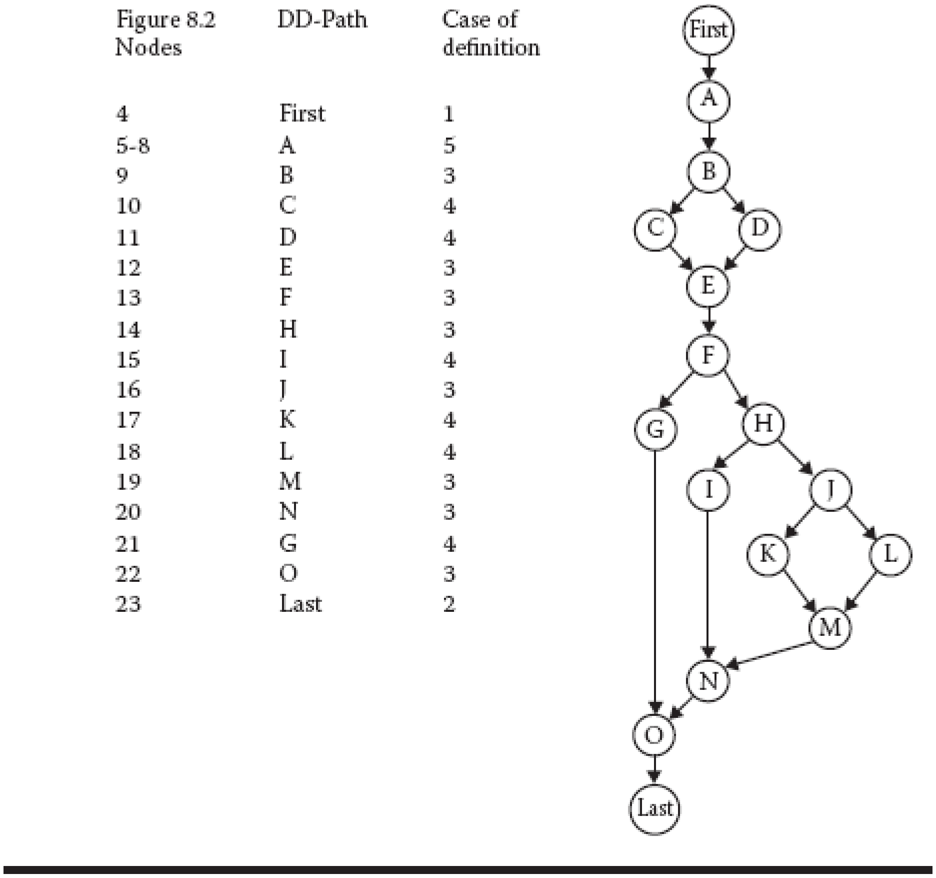


Figure 8.2 Nodes	DD-Path	Case of definition
4	First	1
5-8	A	5
9	B	3
10	C	4
11	D	4
12	E	3
13	F	3
14	H	3
15	I	4
16	J	3
17	K	4
18	L	4
19	M	3
20	N	3
21	G	4
22	O	3
23	Last	2

3.5 DD-path graph for triangle program.

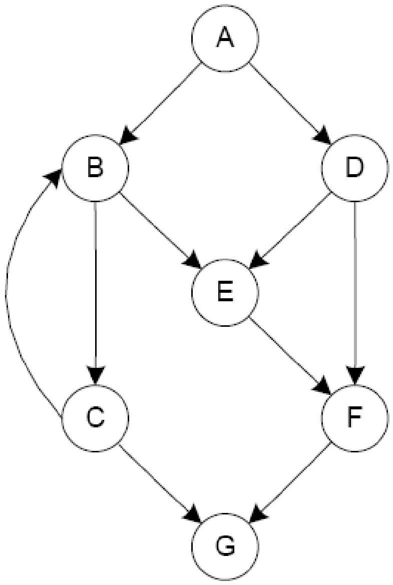
7 b:
Ans:

(McCabe) Basis Path Testing

- in math, a basis "spans" an entire space, such that everything in the space can be derived from the basis elements.
- the cyclomatic number of a strongly connected directed graph is the number of linearly independent cycles.
- given a program graph, we can always add an edge from the sink node to the source node to create a strongly connected graph. (assuming single entry, single exit)
- computing V(G) = e - n + p from the modified program graph yields the number of independent paths that must be tested.
- since all other program execution paths are linear combinations of the basis path, it is necessary to test the basis paths. (Some say this is sufficient; but that is problematic.)
- the next few slides follow McCabe's original example.

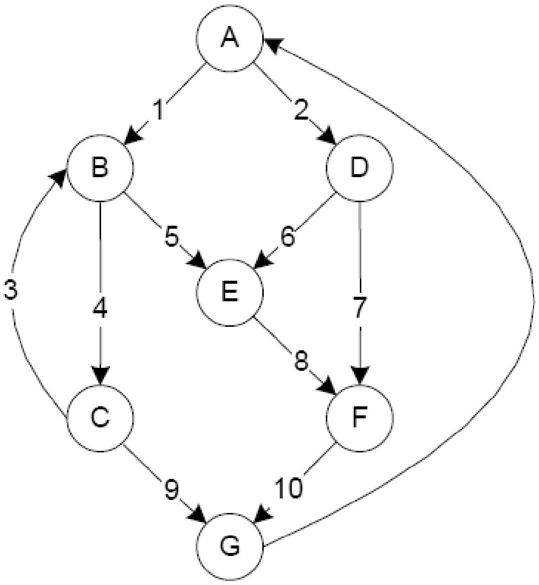
McCabe's Example

McCabe's Original Graph



$$V(G) = 10 - 7 + 2(1)$$
$$= 5$$

Derived, Strongly Connected Graph

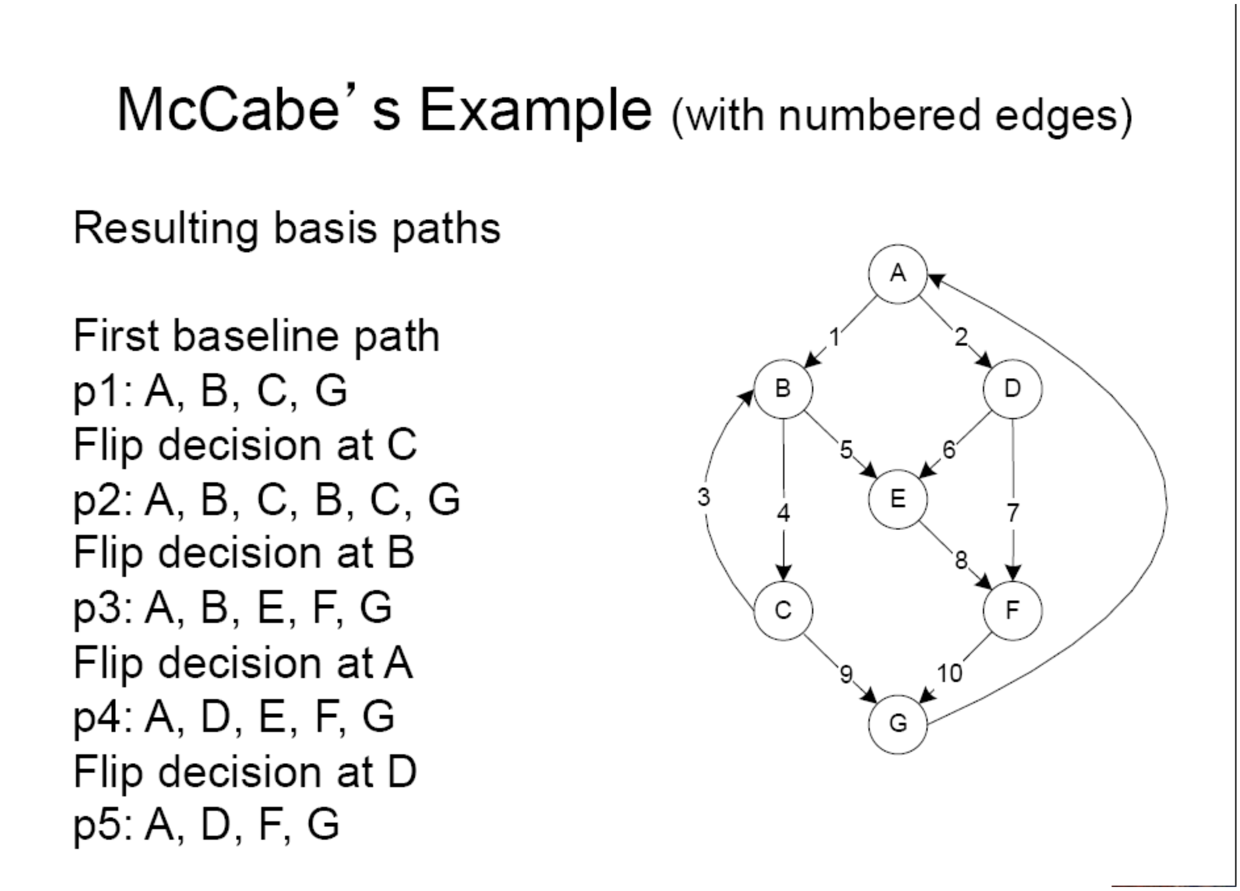


$$V(G) = 11 - 7 + 1$$
$$= 5$$

McCabe's Baseline Method

- Pick a "baseline" path that corresponds to normal execution. (The baseline should have as many decisions as possible.)
- To get succeeding basis paths, retrace the baseline until you reach a decision node. "Flip" the decision (take another alternative) and continue as much of the baseline as possible.
- Repeat this until all decisions have been flipped. When you reach V(G) basis paths, you're done.
- If there aren't enough decisions in the first baseline path, find a second baseline and repeat steps 2 and 3.

Following this algorithm, we get basis paths for McCabe's example.



McCabe's Example (with numbered edges)

Resulting basis paths

First baseline path
p1: A, B, C, G
Flip decision at C
p2: A, B, C, B, C, G
Flip decision at B
p3: A, B, E, F, G
Flip decision at A
p4: A, D, E, F, G
Flip decision at D
p5: A, D, F, G

Q8 a:
Ans:

Define/Use Test Coverage Metrics

The whole point of analyzing a program with definition/use paths is to define a set of test coverage metrics known as the Rapps–Weyuker data flow metrics (Rapps and Weyuker, 1985). The first three of these are equivalent to three of E.F. Miller's metrics in Chapter 8: All-Paths, All-Edges, and All-Nodes. The others presume that define and usage nodes have been identified for all program variables, and that du-paths have been identified with respect to each variable. In the following definitions, *T* is a set of paths in the program graph *G*(*P*) of a program *P*, with the set *V* of variables. It is not enough to take the cross product of the set of DEF nodes with the set of USE nodes for a variable to define du-paths. This mechanical approach can result in infeasible paths. In the next definitions, we assume that the define/use paths are all feasible.

Definition

The set *T* satisfies the *All-Defs criterion* for the program *P* if and only if for every variable $v \in V$, *T* contains definition-clear paths from every defining node of *v* to a use of *v*.

Definition

The set *T* satisfies the *All-Uses criterion* for the program *P* if and only if for every variable $v \in V$, *T* contains definition-clear paths from every defining node of *v* to every use of *v*, and to the successor node of each USE(*v*, *n*).

Definition

The set *T* satisfies the *All-P-Uses/Some C-Uses criterion* for the program *P* if and only if for every variable $v \in V$, *T* contains definition-clear paths from every defining node of *v* to every predicate use of *v*; and if a definition of *v* has no P-uses, a definition-clear path leads to at least one computation use.

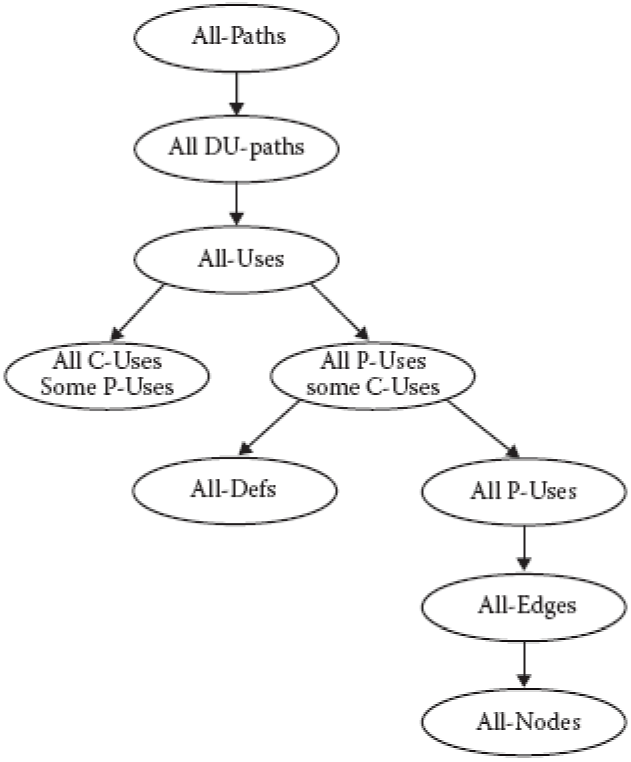
Definition

The set *T* satisfies the *All-C-Uses/Some P-Uses criterion* for the program *P* if and only if for every variable $v \in V$, *T* contains definition clear paths from every defining node of *v* to every computation use of *v*; and if a definition of *v* has no C-uses, a definition-clear path leads to at least one predicate use.

Definition

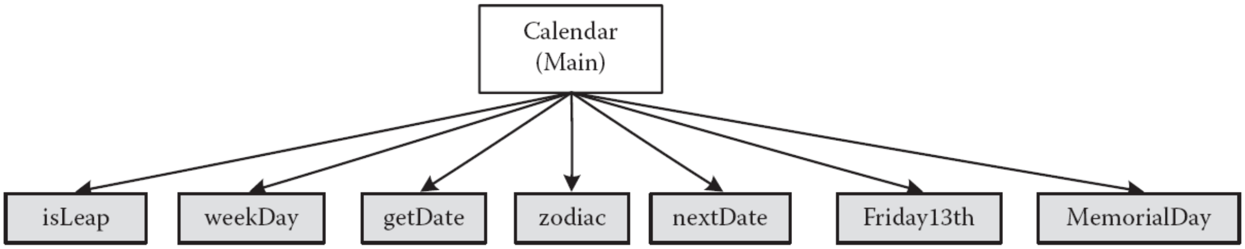
The set *T* satisfies the *All-DU-paths criterion* for the program *P* if and only if for every variable $v \in V$, *T* contains definition-clear paths from every defining node of *v* to every use of *v* and to the successor node of each USE(*v*, *n*), and that these paths are either single loop traversals or they are cycle free.

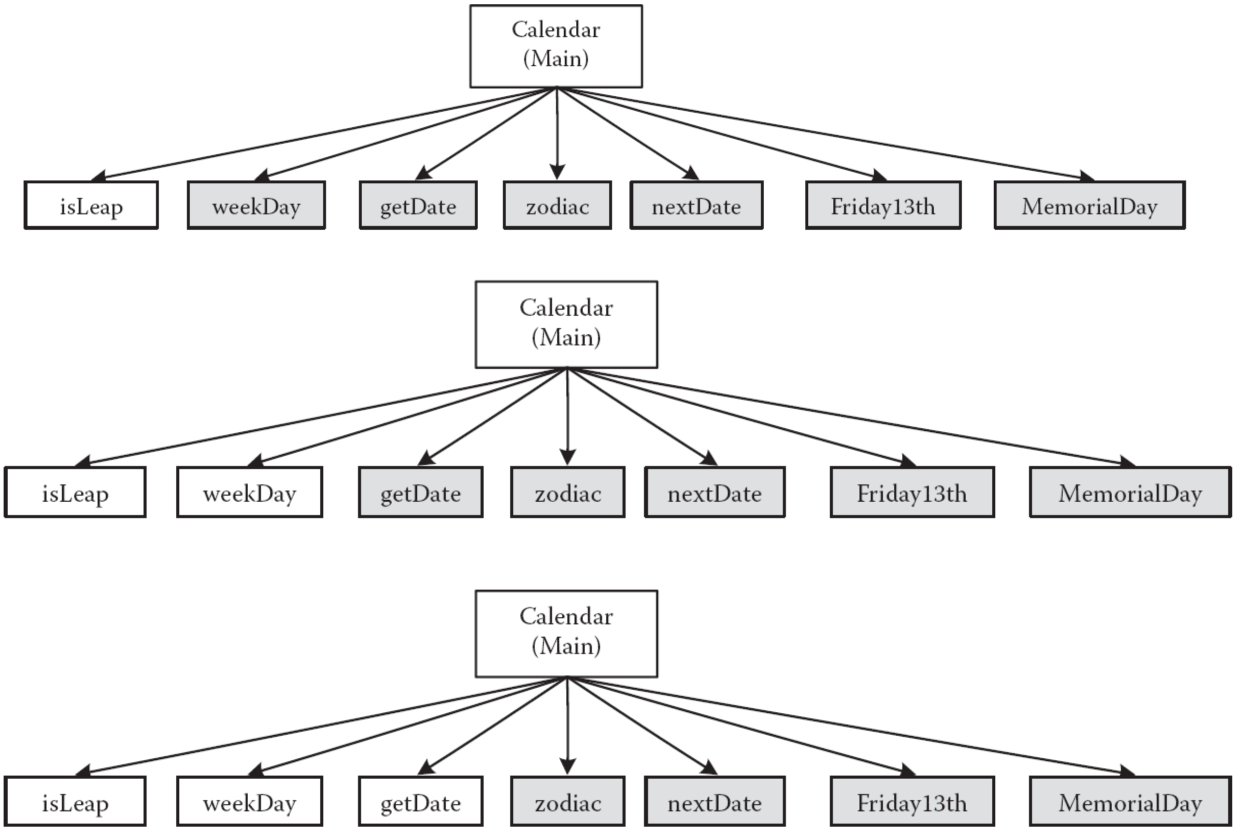
These test coverage metrics have several set-theory-based relationships, which are referred to as "subsumption" in Rapps and Weyuker (1985). These relationships are shown in Figure 9.5. We now have a more refined view of structural testing possibilities between the extremes of the (typically unattainable) All-Paths metric and the generally accepted minimum, All-Edges. What good is all this? Define/use testing provides a rigorous, systematic way to examine points at which faults may occur.



8 b:
Ans:
Top–down integration begins with the main program (the root of the tree). Any lower-level unit that is called by the main program appears as a "stub," where stubs are pieces of throwaway code that emulate a called unit. If we performed top–down integration testing for the Calendar program, the first step would be to develop stubs for all the units called by the main program—isLeap, weekDay, getDate, zodiac, nextDate, friday13th, and memorialDay. In a stub for any unit, the tester hard codes in a correct response to the request from the calling/invoking unit. In the stub for zodiac, for example, if the main program calls zodiac with 05, 27, 2012, zodiacStub would return "Gemini." In extreme practice, the response might be "pretend zodiac returned Gemini."

Once the main program has been tested, we replace one stub at a time, leaving the others as stubs.





Bottom–Up Integration

Bottom–up integration is a “mirror image” to the top–down order, with the difference that stubs are replaced by driver modules that emulate units at the next level up in the tree. (In Figure 13.4, the gray units are drivers.) Bottom–up integration begins with the leaves of the decomposition tree, and use a driver version of the unit that would normally call it to provide it with test cases.

Sandwich Integration
Sandwich integration is a combination of top–down and bottom–up integration.

Q 9 a:
Ans:

Mutation testing (or **mutation analysis** or program **mutation**) is used to design new software tests and evaluate the quality of existing software tests. **Mutation testing** involves modifying a program in small ways.

Mutation analysis is the most common form of software fault-based testing. A fault model is used to produce hypothetical faulty programs by creating variants of the program under test. Variants are created by “seeding” faults, that is, by making a small change to the program under test following a pattern in the fault model. The patterns Δ mutation operator for changing program text are called mutation operators, and each variant program is Δ mutant called a mutant.

Mutants should be plausible as faulty programs. Mutant programs that are rejected by a compiler, or which fail almost all tests, are not good models of the faults we seek Δ valid mutant to uncover with systematic testing. We say a mutant is valid if it is syntactically Δ useful mutant correct. We say a mutant is useful if, in addition to being valid, its behavior differs from the behavior of the original program for no more than a small subset of program test cases.

Fault Based Adequacy Criteria:
Adequacy criteria • Adequacy criterion = set of test obligations • A test suite satisfies an adequacy criterion if – all the tests succeed (pass) – every test obligation in the criterion is satisfied by at least one of the test cases in the test suite. – Example: the statement coverage adequacy criterion is satisfied by test suite S for program P if each executable statement in P is executed by at least one test case in S, and the outcome of each test execution was “pass”.
● Create tests to cover faults that could possibly occur in the software. ● Introduce mutations into the code. ● See if the tests detect the mutations.

9 b:
Ans: How general should scaffolding be? To answer
 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... or further, generate the data automatically from a more abstract model (e.g., network traffic model)
 Fully generic scaffolding may suffice for small numbers of hand-written test cases
 The simplest form of scaffolding is a driver program that runs a single, specific test case.
 It is worthwhile to write more generic test drivers that essentially interpret test case specifications.
 A large suite of automatically generated test cases and a smaller set of handwritten test cases can share the same underlying generic test scaffolding
 Scaffolding to replace portions of the system is somewhat more demanding and again both generic and application-specific approaches are possible
 A simplest stub – ***mock*** – can be generated automatically by analysis of the source code
 The balance of quality, scope and cost for a substantial piece of scaffolding software can be used in several projects
 The balance is altered in favour of simplicity and quick construction for the many small pieces of scaffolding that are typically produced during development to support unit and small-scale integration testing
 A question of costs and re-use – Just as for other kinds of software

Q10 a:

Ans: **Quality Risk Management** is the set of leadership, business **process**, culture, and technology capabilities an organizations establishes to create a collaborative approach for for identifying, quantifying, and mitigating product, operational, supplier, and supply chain **risks** that can impact **quality**.
Risk Planning:

Planning risks and contingencies

What are the overall risks to the project with an emphasis on the testing process?
Lack of personnel resources when testing is to begin.

Lack of availability of required hardware, software, data or tools.

Late delivery of the software, hardware or tools.

Delays in training on the application and/or tools.

Changes to the original requirements or designs.

Complexities involved in testing the applications

Specify what will be done for various events, for example: Requirements definition will be complete by January 1, 20XX, and, if the requirements change after that date, the following actions will be taken:
The test schedule and development schedule will move out an appropriate number of days. This rarely occurs, as most projects tend to have fixed delivery dates.

The number of tests performed will be reduced.

The number of acceptable defects will be increased.

Resources will be added to the test team.

The test team will work overtime (this could affect team morale).

The scope of the plan may be changed.

There may be some optimization of resources. This should be avoided, if possible, for obvious reasons.

10 b:
Ans: Test documentation is documentation of artifacts created before or during the testing of software. It helps the testing team to estimate testing effort needed, test coverage, resource tracking, execution progress, etc. It is a complete suite of documents that allows you to describe and document test planning, test design, test execution, test results that are drawn from the testing activity.

Testing activities generally consume 30% to 50% of software development project effort. Documentations help to identify Test process improvement that can be applied to future projects.

Examples of Test Documentation

Here, are important Types of Test Documentation:

Types of Testing	**Description**
Test policy	It is a high-level document which describes principles, methods and all the important testing goals of the organization.
Test strategy	A high-level document which identifies the Test Levels (types) to be executed for the project.
Test plan	A test plan is a complete planning document which contains the scope, approach, resources, schedule, etc. of testing activities.
Requirements Traceability Matrix	This is a document which connects the requirements to the test cases.
Test Scenario	Test scenario is an item or event of a software system which could be verified by one or more Test cases.
Test case	It is a group of input values, execution preconditions, expected execution postconditions and results. It is developed for a Test Scenario.
Test Data	Test Data is a data which exists before a test is executed. It used to execute the test case.
Defect Report	Defect report is a documented report of any flaw in a Software System which fails to perform its expected function.
Test summary report	Test summary report is a high-level document which summarizes testing activities conducted as well as the test result.

Advantages of Test Documentation

- The main reason behind creating test documentation is to either reduce or remove any uncertainties about the testing activities. Helps you to remove ambiguity which often arises when it comes to the allocation of tasks
- Documentation not only offers a systematic approach to software testing, but it also acts as training material to freshers in the software testing process
- It is also a good marketing &

sales strategy to showcase Test Documentation to exhibit a mature testing process

- Test documentation helps you to offer a quality product to the client within specific time limits
- In Software Engineering, Test Documentation also helps to configure or set-up the program through the configuration document and operator manuals
- Test documentation helps you to improve transparency with the client