CBCS Scheme

16MCA43

USN 1 CY 17 MCA 78

Fourth Semester MCA Degree Examination, June/July 2018
Software Testing and Practices

Time: 3 hrs. Max. Marks: 80

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

Module-1

1 a. Explain error, faults and failures in the process of programming and testing with a diagram. **(10 Marks)**

b. Describe Quality attributes of software testing. **(06 Marks)**

OR

2 a. Explain the basic principles of Analysis and testing and describe adequacy criteria and comparison criteria. **(10 Marks)**

b. Discuss the various test generation strategies in brief. **(06 Marks)**

Module-2

3 a. Describe the implementation of NEXTDate function. **(10 Marks)**

b. Explain the levels of testing found in waterfall model. **(06 Marks)**

OR

4 a. Describe the specified, implemented and tested behavior with the help of Venn diagram and discuss each of them. **(10 Marks)**

b. Write the algorithm for triangle problem. **(06 Marks)**

Module-3

5 a. Illustrate with appropriate diagrams, the mechanism to generate test cases in BVA for a function of two variables in
(i) Robustness Testing
(ii) Worst – Case Testing
(iii) Robust Worst Case Testing. **(10 Marks)**

b. Write the test cases for NEXTDate function using boundary value analysis. **(06 Marks)**

OR

6 a. Write test cases for a commission problem using equivalence class Testing. **(10 Marks)**

b. Illustrate the usage of decision table method to devise test cases for a triangle problem. **(06 Marks)**

Module-4

7 a. Draw a program graph and DD path graph for a triangle problem. **(10 Marks)**

b. Explain McCabe's basis path method with an example. **(06 Marks)**

OR

8 a. Discuss the types of test coverage metrics in detail. **(10 Marks)**

b. Explain traditional view of testing levels and alternative life cycle models. **(06 Marks)**

Module-5

9 a. Explain Scaffolding. Compare between generic versus specific scaffolding. **(07 Marks)**

b. Explain any five major activities in a planning and monitoring the process. **(09 Marks)**

OR

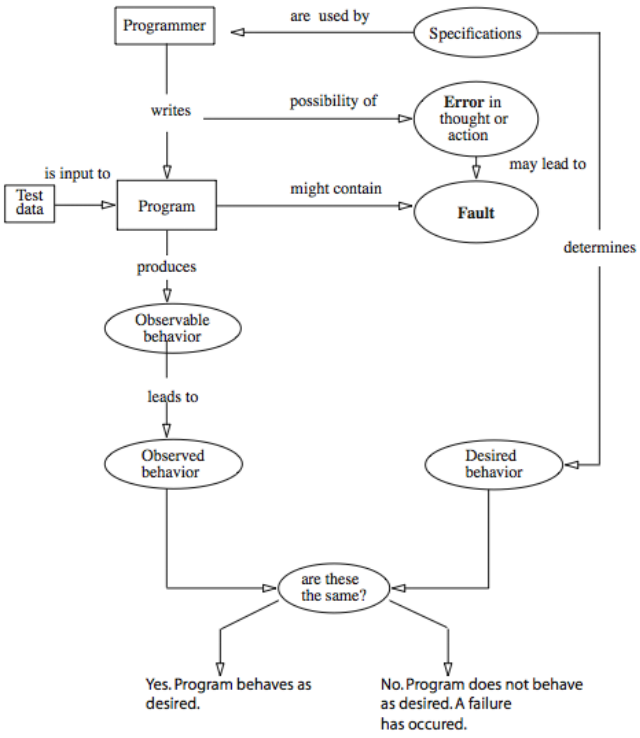
10 a. Describe test oracles and self checks as oracles. **(10 Marks)**

b. Illustrate mutation analysis with its variants. **(06 Marks)**

* * * * *

Solution:

1. A.



Humans make errors in their thoughts, in their actions, and in the products that might result from their actions.

- Humans can make errors in an field.

Ex: observation, in speech, in medical prescription, in surgery, in driving, in sports, in love and similarly even in software development.

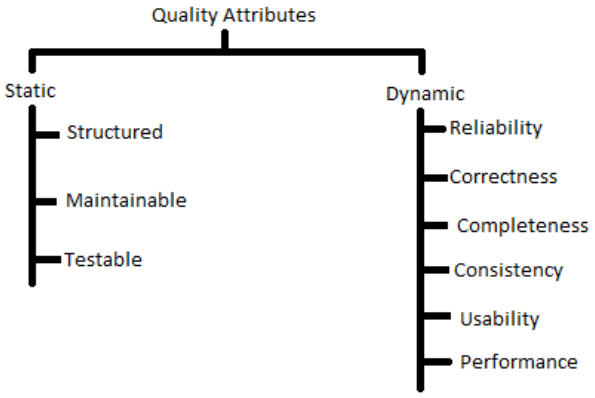
- Example:
  - o An instructor administers a test to determine how well the students have understood what the instructor wanted to convey
  - o A tennis coach administers a test to determine how well the understudy makes a serve

Errors, Faults and Failures Error: An error occurs in the process of writing a program

Fault: a fault is a manifestation of one or more errors

Failure: A failure occurs when a faulty piece of code is executed leading to an incorrect state that propagates to program's output

1. B.



Static quality attributes: structured, maintainable, testable code as well as the availability of correct and complete documentation.
Dynamic quality attributes: software reliability, correctness, completeness, consistency, usability, and performance
Reliability is a statistical approximation to correctness, in the sense that 100% reliability is indistinguishable from correctness. Roughly speaking, reliability is a measure of the likelihood of correct function for some "unit" of behavior, which could be a single use or program execution or a period of time.
Correctness will be established via requirement specification and the program text to prove that software is behaving as expected. Though correctness of a program is desirable, it is almost never the objective of testing. To establish correctness via testing would imply testing a program on all elements in the input domain. In most cases that are encountered in practice, this is impossible to accomplish. Thus correctness is established via mathematical proofs of programs.
While correctness attempts to establish that the program is error free, testing attempts to find if there are any errors in it. Thus completeness of testing does not necessarily demonstrate that a program is error free.
Completeness refers to the availability of all features listed in the requirements, or in the user manual. Incomplete software is one that does not fully implement all features required.
Consistency refers to adherence to a common set of conventions and assumptions. For example, all buttons in the user interface might follow a common color coding convention. An example of inconsistency would be when a database application displays the date of birth of a person in the database.
Usability refers to the ease with which an application can be used. This is an area in itself and there exist techniques for usability testing. Psychology plays an important role in the design of techniques for usability testing.
Performance refers to the time the application takes to perform a requested task. It is considered as a non-functional requirement. It is specified in terms such as ``This task must be performed at the rate of X units of activity in one second on a machine running at speed Y, having Z gigabytes of memory."

2. A. The six basic principles of software testing are:

- General engineering principles:
  - Partition: divide and conquer
  - Visibility: making information accessible
  - Feedback: tuning the development process
- Specific A&T principles:
  - Sensitivity: better to fail every time than sometimes
  - Redundancy: making intentions explicit
  - Restriction: making the problem easier

Partition: Hardware testing and verification problems can be handled by suitably partitioning the input space

Visibility: The ability to measure progress or status against goals. X visibility = ability to judge how we are doing on X, e.g., schedule visibility = "Are we ahead or behind schedule," quality visibility = "Does quality meet our objectives?"

Feedback: The ability to measure progress or status against goals

X visibility = ability to judge how we are doing on X, e.g., schedule visibility = "Are we ahead or behind schedule," quality visibility = "Does quality meet our objectives?"

Sensitivity: A test selection criterion works better if every selected test provides the same result, i.e., if the program fails with one of the selected tests, it fails with all of them (reliable criteria). Run time deadlock analysis works better if it is machine independent, i.e., if the program deadlocks when analyzed on one machine, it deadlocks on every machine

Redundancy: Redundant checks can increase the capabilities of catching specific faults early or more efficiently.

e.g, Static type checking is redundant with respect to dynamic type checking, but it can reveal many type mismatches earlier and more efficiently.

Restriction: Suitable restrictions can reduce hard (unsolvable) problems to simpler (solvable) problems

2. B. **Test Generation Strategies**

Are crucial for the success of the test effort and the accuracy of test plan and estimates

Test generation strategies have the following major tasks:

- Designing the Tests
- Evaluating testability of the requirements and system.

Designing the test environment set-up and identifying any required infrastructure and tools.

Designing the Tests

- Transforms a source document into test designs.
- It involves a set of input values, execution preconditions, expected
results and execution post conditions, developed for a particular objective or test condition, such as to exercise a particular program path or to verify compliance with a specific requirement.

Types of Test Generation Strategies

Analytical Test Generation
• Requirement Based Test Generation
• Risk Based Test Generation
- Code based Test Generation
• Program Mutation
• Control Flow Based Test Generation

Model Based Test Generation

Mathematical Model for critical system behavior

• Requirements are modeled using a formal notations Examples: Finite State Machines, State Charts, Pertrinets, Timed I/O Automata, Algebraic and predicate logic, Sequence and Activity Diagram in UML

Quality Profiling Based Test Generation

Methodical Test Generation

Check list evolved in the organization over years of time which follow industry standard
for software quality

Process or Standard Compliant Based Test Generation

• Agile methodologies such as Xtreme programming
• Industry process or standards like IEEE 829
- Dynamic Test Generation
• Exploratory Testing
- Consultative or Directed Test Generation
• Involving users or developers
Regression - averse Test Generation
• Trying to automate all tests of system functionality prior to release of the function

Selection of any of the strategies depends on risks, skills, objectives, regulations, product,business.

Test generation

Any form of test generation uses a source document.

In the most informal of test methods, the source document resides in the mind of the tester who generates tests based on knowledge of the requirements. In most commercial environments, the process is a bit more formal. The tests are generated using a mix of formal and informal methods either directly from the requirements document serving as the source.

In more advanced test processes, requirements serve as a source for the development of formal models.

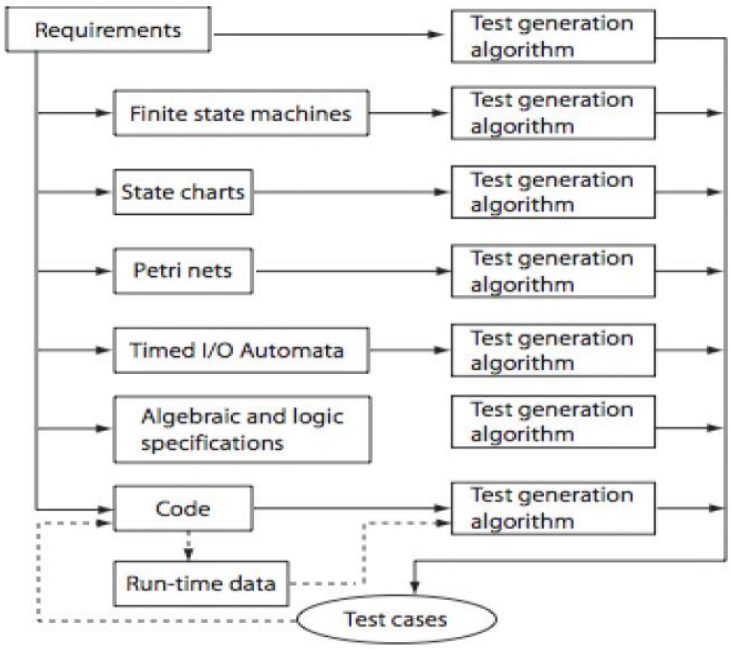
Test generation strategies can be summarized as follows:

Model based: require that a subset of the requirements be modeled using a formal notation (usually graphical). Models: Finite State Machines, Timed automata, Petri net, etc.

Specification based: require that a subset of the requirements be modeled using a formal mathematical notation.

Code based: generate tests directly from the code

Test generation strategies (Summary)



3. A.

NextDate is a function of three variables: month, date, and year. It returns the date of the day after the input date. The month, date, and year variables have integer values subject to these conditions (the year range ending in 2012 is arbitrary, and is from the first edition):
c1. $1 \leq \text{month} \leq 12$
c2. $1 \leq \text{day} \leq 31$
c3. $1812 \leq \text{year} \leq 2012$
As we did with the triangle program, we can make our problem statement more specific. This entails defining responses for invalid values of the input values for the day, month, and year. We can also define responses for invalid combinations of inputs, such as June 31 of any year. If any of conditions c1, c2, or c3 fails, NextDate produces an output indicating the corresponding variable has an out-of-range value—for example, "Value of month not in the range 1...12." Because numerous invalid day–month–year combinations exist, NextDate collapses these into one message: "Invalid Input Date."

Two sources of complexity exist in the NextDate function: the complexity of the input domain discussed previously, and the rule that determines when a year is a leap year. A year is 365.2422 days long; therefore, leap years are used for the "extra day" problem. If we declared a leap year every fourth year, a slight error would occur. The Gregorian calendar (after Pope Gregory) resolves this by adjusting leap years on century years. Thus, a year is a leap year if it is divisible by 4, unless it is a century year. Century years are leap years only if they are multiples of 400 (Inglis, 1961);

thus, 1992, 1996, and 2000 are leap years, while the year 1900 is not a leap year. The NextDate function also illustrates a sidelight of software testing. Many times, we find examples of Zipf's law,
which states that 80% of the activity occurs in 20% of the space. Notice how much of the source code is devoted to leap year considerations. In the second implementation, notice how much code is devoted to input value validation.

Implementation:

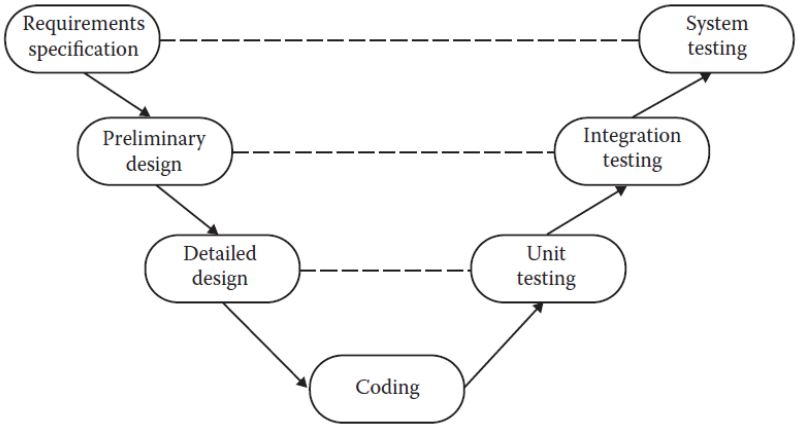
```
Program NextDate1 'Simple version
Dim tomorrowDay,tomorrowMonth,tomorrowYear As Integer
Dim day,month,year As Integer
Output ("Enter today's date in the form MM DD YYYY")
Input (month, day, year)
Case month Of
Case 1: month Is 1,3,5,7,8, Or 10: '31 day months (except Dec.)
        If day < 31
                Then tomorrowDay = day + 1
        Else
                tomorrowDay = 1
                tomorrowMonth = month + 1
        EndIf
Case 2: month Is 4,6,9, Or 11 '30 day months
        If day < 30
                Then tomorrowDay = day + 1
        Else
                tomorrowDay = 1
                tomorrowMonth = month + 1
        EndIf
Case 3: month Is 12: 'December
        If day < 31
                Then tomorrowDay = day + 1
        Else
                tomorrowDay = 1
                tomorrowMonth = 1
        If year = 2012
                Then Output ("2012 is over")
        Else     tomorrow.year = year + 1
        EndIf
Case 4: month is 2: 'February
        If day < 28
                Then tomorrowDay = day + 1
        Else
                If day = 28
                        Then If ((year is a leap year)
                                Then tomorrowDay = 29 'leap year
                        Else 'not a leap year
                                tomorrowDay = 1
                                tomorrowMonth = 3
                        EndIf
                Else If day = 29
                        Then If ((year is a leap year)
                                Then tomorrowDay = 1tomorrowMonth = 3
                        Else 'not a leap year
                                Output("Cannot have Feb.", day)
                        EndIf
                EndIf
        EndIf
EndIf
EndCase
Output ("Tomorrow's date is", tomorrowMonth, tomorrowDay, tomorrowYear)
End NextDate
```

Program NextDate2 Improved version

```
‘
Dim tomorrowDay,tomorrowMonth,tomorrowYear As Integer
Dim day,month,year As Integer
Dim c1, c2, c3 As Boolean
‘
Do
Output (“Enter today’s date in the form MM DD YYYY”)
Input (month, day, year)
c1 = (1 ≤ day) AND (day ≤ 31)
c2 = (1 ≤ month) AND (month ≤ 12)
c3 = (1812 ≤ year) AND (year ≤ 2012)
If NOT(c1)
        Then Output(“Value of day not in the range 1..31”)
EndIf
If NOT(c2)
        Then Output(“Value of month not in the range 1..12”)
EndIf
If NOT(c3)
        Then Output(“Value of year not in the range 1812..2012”)
EndIf
Until c1 AND c2 AND c2
Case month Of
Case 1: month Is 1,3,5,7,8, Or 10: ‘31 day months (except Dec.)
If day < 31
        Then tomorrowDay = day + 1
Else
        tomorrowDay = 1
        tomorrowMonth = month + 1
EndIf
Case 2: month Is 4,6,9, Or 11 ‘30 day months
If day < 30
        Then tomorrowDay = day + 1
Else
        If day = 30
                Then tomorrowDay = 1
                tomorrowMonth = month + 1
        Else Output(“Invalid Input Date”)
        EndIf
EndIf
Case 3: month Is 12: ‘December
If day < 31
        Then tomorrowDay = day + 1
Else
        tomorrowDay = 1
        tomorrowMonth = 1
        If year = 2012
                Then Output (“Invalid Input Date”)
        Else tomorrow.year = year + 1
        EndIf
EndIf
Case 4: month is 2: ‘February
If day < 28
        Then tomorrowDay = day + 1
Else
If day = 28
        Then
If (year is a leap year)
        Then tomorrowDay = 29 ‘leap day
Else ‘not a leap year
        tomorrowDay = 1
        tomorrowMonth = 3
EndIf
Else
        If day = 29
```

```
                Then
                If (year is a leap year)
                        Then tomorrowDay = 1
                        tomorrowMonth = 3
        Else
                If day > 29
                        Then Output("Invalid Input Date")
                EndIf
        EndIf
        EndIf
EndIf
EndIf
EndCase
Output ("Tomorrow's date is", tomorrowMonth, tomorrowDay, tomorrowYear)
'
End NextDate2
```

3. B



.

Levels of testing echo the levels of abstraction found in the waterfall model of the software development life cycle. Although this model has its drawbacks, it is useful for testing as a means of identifying distinct levels of testing and for clarifying the objectives that pertain to each level. A diagrammatic variation of the waterfall model, known as the V-Model in ISTQB parlance, is given in Figure 1.8; this variation emphasizes the correspondence between testing and design levels. Notice that, especially in terms of specification-based testing, the three levels of definition (specification, preliminary design, and detailed design) correspond directly to three levels of testing— system, integration, and unit testing. A practical relationship exists between levels of testing versus specification-based and code based testing. Most practitioners agree that code-based testing is most appropriate at the unit level, whereas specification-based testing is most appropriate at the system level. This is generally true; however, it is also a likely consequence of the base information produced during the requirements

specification, preliminary design, and detailed design phases. The constructs defined for code-based testing make the most sense at the unit level, and similar constructs are only now becoming available for the integration and system levels of testing. We develop such structures in Chapters 11 through 17 to support code-based testing at the integration and system levels for both traditional and object-oriented software.

4. A.

Testing is fundamentally concerned with behavior, and behavior is orthogonal to the code-based view common to software (and system) developers. A quick distinction is that the code-based view focuses on what it *is* and the behavioral view considers what it *does*. One of the continuing sources of difficulty for testers is that the base documents are usually written by and for developers; the emphasis is therefore on code-based, instead of behavioral, information. In this section, we develop a simple Venn diagram that clarifies several nagging questions about testing. Consider a universe of program behaviors. (Notice that we are forcing attention on the essence of testing.) Given a program and its specification, consider the set *S* of specified behaviors and the set *P* of programmed behaviors. Figure 1.2 shows the relationship between the specified and programmed behaviors. Of all the possible program behaviors, the specified ones are in the circle

labeled *S* and all those behaviors actually programmed are in *P*. With this diagram, we can see more clearly the problems that confront a tester. What if certain specified behaviors have not been programmed? In our earlier terminology, these are faults of omission. Similarly, what if certain programmed (implemented) behaviors have not been specified? These correspond to faults of commission and to errors that occurred after the specification was complete. The intersection of *S* and *P* (the football-shaped region) is the "correct" portion, that is, behaviors that are both specified and implemented. A very good view of testing is that it is the determination of the extent of program behavior that is both specified and implemented. (As an aside, note that "correctness" only has meaning with respect to a specification and an implementation. It is a relative term, not an absolute.)

The new circle in Figure 1.3 is for test cases. Notice the slight discrepancy with our universe of discourse and the set of program behaviors. Because a test case causes a program behavior, the mathematicians might forgive us. Now, consider the relationships among sets *S*, *P*, and *T*. There may be specified behaviors that are not tested (regions 2 and 5), specified behaviors that are tested (regions 1 and 4), and test cases that correspond to unspecified behaviors (regions 3 and 7). Similarly, there may be programmed behaviors that are not tested (regions 2 and 6), programmed behaviors that are tested (regions 1 and 3), and test cases that correspond to behaviors that were not implemented (regions 4 and 7). Each of these regions is important. If specified behaviors exist for which no test cases are available, the testing is necessarily incomplete. If certain test cases correspond to unspecified behaviors, some possibilities arise: either such a test case is unwarranted, the specification is deficient, or the tester wishes to determine that specified non-behavior does not occur. (In my experience, good testers often postulate test cases of this latter type. This is a fine reason to have good testers participate in specification and design reviews.) We are already at a point where we can see some possibilities for testing as a craft: what can a tester do to make the region where these sets all intersect (region 1) as large as possible? Another approach is to ask how the test cases in set *T* are identified. The short answer is that test cases are identified by a testing method.

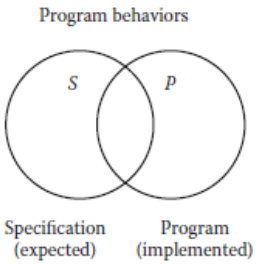


Figure 1.2 Specified and implemented program behaviors.

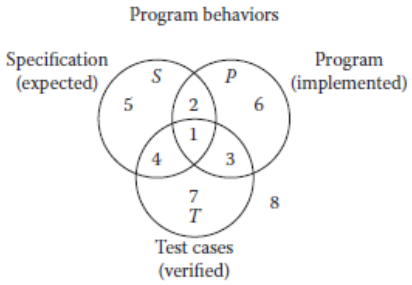


Figure 1.3 Specified, implemented, and tested behaviors.

4. B. Triangle Problem:

```
Dim a, b, c As Integer
Dim c1, c2, c3, IsATriangle As Boolean
'Step 1: Get Input
Do
        Output("Enter 3 integers which are sides of a triangle")
        Input(a, b, c)
        c1 = (1 ≤ a) AND (a ≤ 300)
        c2 = (1 ≤ b) AND (b ≤ 300)
        c3 = (1 ≤ c) AND (c ≤ 300)
        If NOT(c1)
                Then Output("Value of a is not in the range of permitted values")
        EndIf
        If NOT(c2)
                Then Output("Value of b is not in the range of permitted values")
        EndIf
        If NOT(c3)
```

```
            ThenOutput("Value of c is not in the range of permitted values")
        EndIf
Until c1 AND c2 AND c3
Output("Side A is",a)
Output("Side B is",b)
Output("Side C is",c)
'Step 2: Is A Triangle?
If (a < b + c) AND (b < a + c) AND (c < a + b)
        Then IsATriangle = True
Else IsATriangle = False

EndIf
'Step 3: Determine Triangle Type
If IsATriangle
        Then If (a = b) AND (b = c)
                Then Output ("Equilateral")
        Else If (a ≠ b) AND (a ≠ c) AND (b ≠ c)
                Then Output ("Scalene")
        Else Output ("Isosceles")
        EndIf
EndIf
Else Output("Not a Triangle")
EndIf
End triangle3
```

5. A.

   i. Robustness Testing

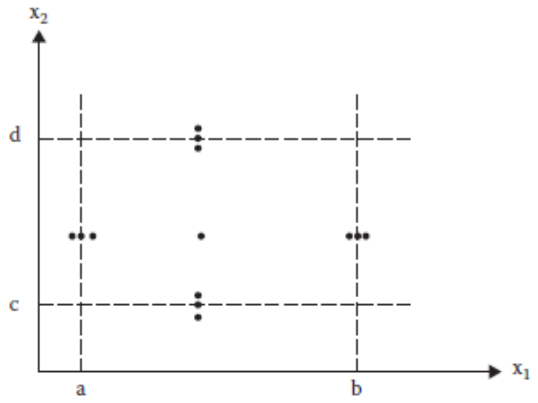


Figure 5.3 Robustness test cases for a function of two variables.

   ii. Worst Case Testing

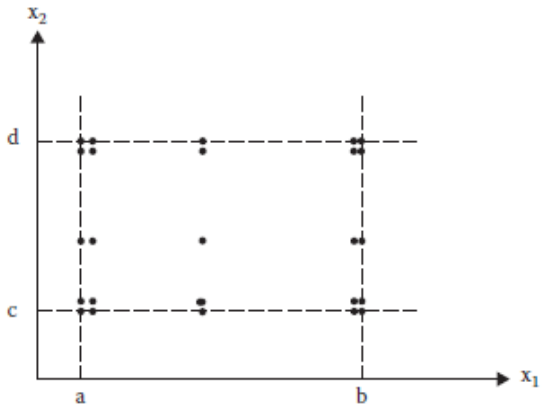


Figure 5.4 Worst-case test cases for a function of two variables.

   iii. Robust Worst Case Testing

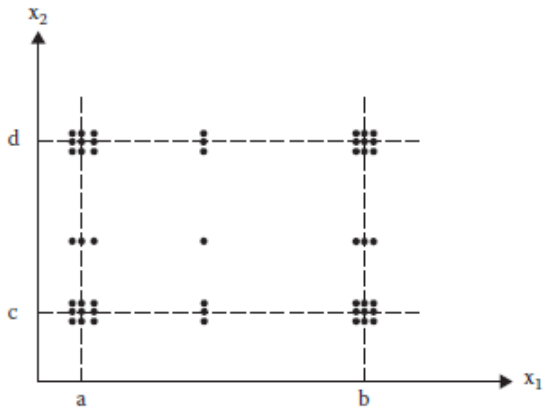


Figure 5.5 Robust worst-case test cases for a function of two variables.

5. B

Table 5.3 Worst-Case Test Cases

Case	*Month*	*Day*	*Year*	*Expected Output*
1	1	1	1812	1, 2, 1812
2	1	1	1813	1, 2, 1813
3	1	1	1912	1, 2, 1912
4	1	1	2011	1, 2, 2011
5	1	1	2012	1, 2, 2012
6	1	2	1812	1, 3, 1812
7	1	2	1813	1, 3, 1813
8	1	2	1912	1, 3, 1912
9	1	2	2011	1, 3, 2011
10	1	2	2012	1, 3, 2012
11	1	15	1812	1, 16, 1812
12	1	15	1813	1, 16, 1813
13	1	15	1912	1, 16, 1912
14	1	15	2011	1, 16, 2011
15	1	15	2012	1, 16, 2012
16	1	30	1812	1, 31, 1812
17	1	30	1813	1, 31, 1813
18	1	30	1912	1, 31, 1912
19	1	30	2011	1, 31, 2011
20	1	30	2012	1, 31, 2012
21	1	31	1812	2, 1, 1812
22	1	31	1813	2, 1, 1813
23	1	31	1912	2, 1, 1912
24	1	31	2011	2, 1, 2011
25	1	31	2012	2, 1, 2012
26	2	1	1812	2, 2, 1812
27	2	1	1813	2, 2, 1813
28	2	1	1912	2, 2, 1912

(continued)

.

Table 5.3 Worst-Case Test Cases (Continued)

Case	*Month*	*Day*	*Year*	*Expected Output*
29	2	1	2011	2, 2, 2011
30	2	1	2012	2, 2, 2012
31	2	2	1812	2, 3, 1812
32	2	2	1813	2, 3, 1813
33	2	2	1912	2, 3, 1912
34	2	2	2011	2, 3, 2011
35	2	2	2012	2, 3, 2012
36	2	15	1812	2, 16, 1812
37	2	15	1813	2, 16, 1813
38	2	15	1912	2, 16, 1912
39	2	15	2011	2, 16, 2011
40	2	15	2012	2, 16, 2012
41	2	30	1812	Invalid date
42	2	30	1813	Invalid date
43	2	30	1912	Invalid date
44	2	30	2011	Invalid date
45	2	30	2012	Invalid date
46	2	31	1812	Invalid date
47	2	31	1813	Invalid date
48	2	31	1912	Invalid date
49	2	31	2011	Invalid date
50	2	31	2012	Invalid date
51	6	1	1812	6, 2, 1812
52	6	1	1813	6, 2, 1813
53	6	1	1912	6, 2, 1912
54	6	1	2011	6, 2, 2011
55	6	1	2012	6, 2, 2012
56	6	2	1812	6, 3, 1812
57	6	2	1813	6, 3, 1813

(*continued*)

Table 5.3 Worst-Case Test Cases (Continued)

Case	*Month*	*Day*	*Year*	*Expected Output*
58	6	2	1912	6, 3, 1912
59	6	2	2011	6, 3, 2011
60	6	2	2012	6, 3, 2012
61	6	15	1812	6, 16, 1812
62	6	15	1813	6, 16, 1813
63	6	15	1912	6, 16, 1912
64	6	15	2011	6, 16, 2011
65	6	15	2012	6, 16, 2012
66	6	30	1812	7, 1, 1812
67	6	30	1813	7, 1, 1813
68	6	30	1912	7, 1, 1912
69	6	30	2011	7, 1, 2011
70	6	30	2012	7, 1, 2012
71	6	31	1812	Invalid date
72	6	31	1813	Invalid date
73	6	31	1912	Invalid date
74	6	31	2011	Invalid date
75	6	31	2012	Invalid date
76	11	1	1812	11, 2, 1812
77	11	1	1813	11, 2, 1813
78	11	1	1912	11, 2, 1912
79	11	1	2011	11, 2, 2011
80	11	1	2012	11, 2, 2012
81	11	2	1812	11, 3, 1812
82	11	2	1813	11, 3, 1813
83	11	2	1912	11, 3, 1912
84	11	2	2011	11, 3, 2011
85	11	2	2012	11, 3, 2012
86	11	15	1812	11, 16, 1812

(continued)

Table 5.3 Worst-Case Test Cases (Continued)

Case	*Month*	*Day*	*Year*	*Expected Output*
87	11	15	1813	11, 16, 1813
88	11	15	1912	11, 16, 1912
89	11	15	2011	11, 16, 2011
90	11	15	2012	11, 16, 2012
91	11	30	1812	12, 1, 1812
92	11	30	1813	12, 1, 1813
93	11	30	1912	12, 1, 1912
94	11	30	2011	12, 1, 2011
95	11	30	2012	12, 1, 2012
96	11	31	1812	Invalid date
97	11	31	1813	Invalid date
98	11	31	1912	Invalid date
99	11	31	2011	Invalid date
100	11	31	2012	Invalid date
101	12	1	1812	12, 2, 1812
102	12	1	1813	12, 2, 1813
103	12	1	1912	12, 2, 1912
104	12	1	2011	12, 2, 2011
105	12	1	2012	12, 2, 2012
106	12	2	1812	12, 3, 1812
107	12	2	1813	12, 3, 1813
108	12	2	1912	12, 3, 1912
109	12	2	2011	12, 3, 2011
110	12	2	2012	12, 3, 2012
111	12	15	1812	12, 16, 1812
112	12	15	1813	12, 16, 1813
113	12	15	1912	12, 16, 1912
114	12	15	2011	12, 16, 2011
115	12	15	2012	12, 16, 2012

(*continued*)

Table 5.3 Worst-Case Test Cases (Continued)

Case	*Month*	*Day*	*Year*	*Expected Output*
116	12	30	1812	12, 31, 1812
117	12	30	1813	12, 31, 1813
118	12	30	1912	12, 31, 1912
119	12	30	2011	12, 31, 2011
120	12	30	2012	12, 31, 2012
121	12	31	1812	1, 1, 1813
122	12	31	1813	1, 1, 1814
123	12	31	1912	1, 1, 1913
124	12	31	2011	1, 1, 2012
125	12	31	2012	1, 1, 2013

6. A.

Case ID	*Locks*	*Stocks*	*Barrels*	*Expected Output*
WR1	10	10	10	$100
WR2	–1	40	45	Program terminates
WR3	–2	40	45	Value of locks not in the range 1 ... 70
WR4	71	40	45	Value of locks not in the range 1 ... 70
WR5	35	–1	45	Value of stocks not in the range 1 ... 80
WR6	35	81	45	Value of stocks not in the range 1 ... 80
WR7	35	40	–1	Value of barrels not in the range 1 ... 90
WR8	35	40	91	Value of barrels not in the range 1 ... 90

Case ID	*Locks*	*Stocks*	*Barrels*	*Expected Output*
SR1	–2	40	45	Value of locks not in the range 1 ... 70
SR2	35	–1	45	Value of stocks not in the range 1 ... 80
SR3	35	40	–2	Value of barrels not in the range 1 ... 90
SR4	–2	–1	45	Value of locks not in the range 1 ... 70 Value of stocks not in the range 1 ... 80
SR5	–2	40	–1	Value of locks not in the range 1 ... 70 Value of barrels not in the range 1 ... 90
SR6	35	–1	–1	Value of stocks not in the range 1 ... 80 Value of barrels not in the range 1 ... 90
SR7	–2	–1	–1	Value of locks not in the range 1 ... 70 Value of stocks not in the range 1 ... 80 Value of barrels not in the range 1 ... 90

Test Case	*Locks*	*Stocks*	*Barrels*	*Sales*	*Commission*
OR1	5	5	5	500	50
OR2	15	15	15	1500	175
OR3	25	25	25	2500	360

6. B.

Table 7.5 Decision Table for Table 7.3 with Rule Counts

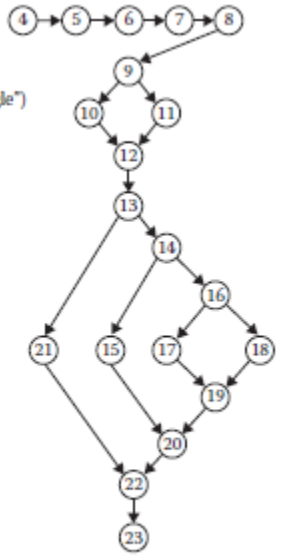
c1: a < b + c?	F	T	T	T	T	T	T	T	T	T	T
c2: b < a + c?	—	F	T	T	T	T	T	T	T	T	T
c3: c < a + b?	—	—	F	T	T	T	T	T	T	T	T
c4: a = b?	—	—	—	T	T	T	T	F	F	F	F
c5: a = c?	—	—	—	T	T	F	F	T	T	F	F
c6: b = c?	—	—	—	T	F	T	F	T	F	T	F
Rule count	32	16	8	1	1	1	1	1	1	1	1
a1: Not a triangle	X	X	X								
a2: Scalene											X
a3: Isosceles							X		X	X	
a4: Equilateral				X							
a5: Impossible					X	X		X			

Table 7.11 Test Cases from Table 7.3

Case ID	a	b	c	*Expected Output*
DT1	4	1	2	Not a triangle
DT2	1	4	2	Not a triangle
DT3	1	2	4	Not a triangle
DT4	5	5	5	Equilateral
DT5	?	?	?	Impossible
DT6	?	?	?	Impossible
DT7	2	2	3	Isosceles
DT8	?	?	?	Impossible
DT9	2	3	2	Isosceles
DT10	3	2	2	Isosceles
DT11	3	4	5	Scalene

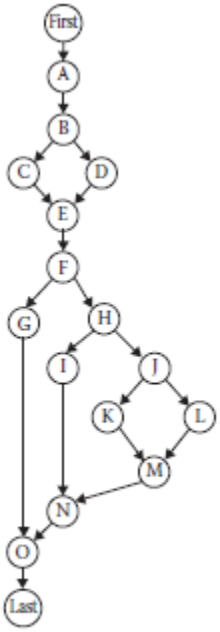
7. A.

```
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
```



Program graph of triangle program.

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



DD-path graph for triangle program.

7. B. Figure 8.10 is taken from McCabe (1982). It is a directed graph that we might take to be the program graph (or the DD-path graph) of some program. For the convenience of readers who have encountered this example elsewhere (McCabe, 1987; Perry, 1987), the original notation for nodes and edges is repeated here. (Notice that this is not a graph derived from a structured program:
nodes B and C are a loop with two exits, and the edge from B to E is a branch into the if–then statement in nodes D, E, and F.) The program does have a single entry (A) and a single exit (G). McCabe based his view of testing on a major result from graph theory, which states that the cyclomatic number (see Chapter 4) of a strongly connected graph is the number of linearly independent circuits in the graph. (A circuit is similar to a chain: no internal loops or decisions occur, but the initial node is the terminal node. A circuit is a set of 3-connected nodes.) We can always create a strongly connected graph by adding an edge from the (every) sink node to the (every) source node. (Notice that, if the single-entry, single-exit precept is violated, we greatly increase the cyclomatic number because we need to add edges from each sink node to each source node.) The right side of Figure 8.10 shows the result of doing this; it also contains edge labels that are used in the discussion that follows. Some confusion exists in the literature about the correct formula for cyclomatic complexity. Some sources give the formula as $V(G) = e - n + p$, while others use the formula $V(G) = e - n + 2p$; everyone agrees that e is the number of edges, n is the number of nodes, and p is the number of connected regions. The confusion apparently comes from the transformation of an arbitrary directed graph (such as the one in Figure 8.10, left side) to a strongly connected, directed graph obtained by adding one edge from the sink to the source node (as in Figure 8.10, right side). Adding an edge clearly affects value computed by the formula, but it should not affect the number of circuits. Counting or not counting the added edge accounts for the change to the coefficient of p, the number of connected regions. Since p is usually 1, adding the extra edge means we move from $2p$ to p. Here is a way to resolve the apparent inconsistency. The number of linearly independent paths from the source node to the sink node of the graph on the left side of Figure 8.10 is

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

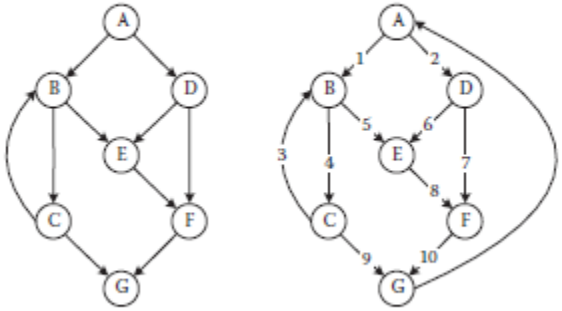


Figure 8.10 McCabe's control graph and derived strongly connected graph.

The number of linearly independent circuits of the graph on the right side of the graph in Figure 8.10 is

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

The cyclomatic complexity of the strongly connected graph in Figure 8.10 is 5; thus, there are five linearly independent circuits. If we now delete the added edge from node G to node A, these five circuits become five linearly independent paths from node A to node G. In small graphs, we can visually identify independent paths. Here, we identify paths as sequences of nodes:
p1: A, B, C, G
p2: A, B, C, B, C, G
p3: A, B, E, F, G
p4: A, D, E, F, G
p5: A, D, F, G
Table 8.5 shows the edges traversed by each path, and also the number of times an edge is traversed. We can force this to begin to look like a vector space by defining notions of addition and scalar multiplication: path addition is simply one path followed by another path, and multiplication corresponds to repetitions of a path. With this formulation, McCabe arrives at a vector space of program paths. His illustration of the basis part of this framework is that the path A, B, C, B, E, F, G is the basis sum p2 + p3 – p1, and the path A, B, C, B, C, B, C, G is the linear combination 2p2 – p1. It is easier to see this addition with an incidence matrix (see Chapter 4) in which rows
correspond to paths, and columns correspond to edges, as in Table 8.5. The entries in this table are obtained by following a path and noting which edges are traversed. Path p1, for example, traverses edges 1, 4, and 9, while path p2 traverses the following edge sequence: 1, 4, 3, 4, 9. Because edge 4 is traversed twice by path p2, that is the entry for the edge 4 column. We can check the independence of paths p1 – p5 by examining the first five rows of this incidence matrix. The bold entries show edges that appear in exactly one path, so paths p2 – p5 must be independent. Path p1 is independent of all of these, because any attempt to express p1 in terms of the others introduces unwanted edges. None can be deleted, and these five paths span the set of all paths from node A to node G. At this point, you might check the linear combinations of the two example paths. (The addition and multiplication are performed on the column entries.) McCabe next develops an algorithmic procedure (called the baseline method) to determine a set of basis paths. The method begins with the selection of a baseline path, which should correspond to some "normal case" program execution. This can be somewhat arbitrary; McCabe advises choosing a path with as many decision nodes as possible. Next, the baseline path is retraced, and in turn each decision is "flipped"; that is, when a node of outdegree ≥ 2 is reached, a different edge must be taken. Here we follow McCabe's example, in which he first postulates the path

through nodes A, B, C, B, E, F, G as the baseline. (This was expressed in terms of paths p1 – p5 earlier.) The first decision node (outdegree ≥ 2) in this path is node A; thus, for the next basis path, we traverse edge 2 instead of edge 1. We get the path A, D, E, F, G, where we retrace nodes E, F, G in path 1 to be as minimally different as possible. For the next path, we can follow the second path, and take the other decision outcome of node D, which gives us the path A, D, F, G. Now, only decision nodes B and C have not been flipped; doing so yields the last two basis paths, A, B, E, F, G and A, B, C, G. Notice that this set of basis paths is distinct from the one in Table 8.6: this is not problematic because a unique basis is not required.

Table 8.5 Path/Edge Traversal

Path/Edges Traversed	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*
p1: A, B, C, G	1	0	0	1	0	0	0	0	1	0
p2: A, B, C, B, C, G	1	0	1	2	0	0	0	0	1	0
p3: A, B, E, F, G	1	0	0	0	1	0	0	1	0	1
p4: A, D, E, F, G	0	1	0	0	0	1	0	1	0	1
p5: A, D, F, G	0	1	0	0	0	0	1	0	0	1
ex1: A, B, C, B, E, F, G	1	0	1	1	1	0	0	1	0	1
ex2: A, B, C, B, C, B, C, G	1	0	2	3	0	0	0	0	1	0

Table 8.6 Basis Paths in Figure 8.5

Original	p1: A–B–C–E–F–H–J–K–M–N–O–Last	Scalene
Flip p1 at B	p2: A–B–D–E–F–H–J–K–M–N–O–Last	Infeasible
Flip p1 at F	p3: A–B–C–E–F–G–O–Last	Infeasible
Flip p1 at H	p4: A–B–C–E–F–H–I–N–O–Last	Equilateral
Flip p1 at J	p5: A–B–C–E–F–H–J–L–M–N–O–Last	Isosceles

8. A.

Fundamental limitations of specification-based testing is that it is impossible to know either the extent of redundancy or the possibility of gaps corresponding to the way a set of functional test cases exercises a program. Test coverage metrics are a device to measure the extent to which a set of test cases covers (or exercises) a program.

Program Graph–Based Coverage Metrics: Given a set of test cases for a program, they constitute *node coverage* if, when executed on the program, every node in the program graph is traversed. Denote this level of coverage as *G*node, where the *G* stands for program graph. Since nodes correspond to statement fragments, this guarantees that every statement fragment is executed by some test case. If we are careful about defining statement fragment nodes, this also guarantees that statement fragments that are outcomes of a decision-making statement are executed.

E.F. Miller's Coverage Metrics: Having an organized view of the extent to which a program is tested makes it possible to sensibly manage the testing process. Most quality organizations now expect the *C*1 metric (DD-path coverage) as the minimum acceptable level of test coverage. These coverage metrics form a lattice in which some are equivalent and some are implied by others. The importance of the lattice is that there are always fault types that can be revealed at one level and can escape detection by inferior levels of testing. Miller (1991) observes that when DD-path coverage is attained by a set of test cases, roughly 85% of all faults are revealed. The test coverage metrics tell us what to test but not how to test it. In this section, we take a closer look at techniques that exercise source code. We must keep an important distinction in mind: Miller's test coverage metrics are based on program graphs in which nodes are full statements, whereas our formulation allows statement fragments (which can be entire statements) to be nodes.

Miller's Test Coverage Metrics

Metric	*Description of Coverage*
C_0	Every statement
C_1	Every DD-path
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

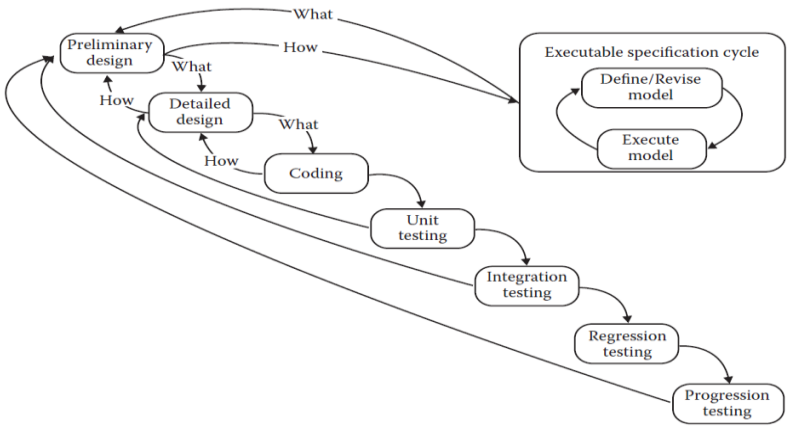
8. B. . **Waterfall Spin Off**

- Development in stages
  - Level use of staff across all types
  - Testing now entails both
    - Regression
    - Progression
- Main variations involve constructing a sequence of systems
  - Incremental
  - Evolutionary
  - Spiral
- Waterfall model is applied to each build
  - Smaller problem than original
  - System functionality does not change
- Incremental
  - Have high-level design at the beginning
  - Low-level design results in a series of builds
    - Incremental testing is useful
  - System testing is not affected
  - Level off staffing problems
- Evolutionary
  - First build is defined
  - Priorities and customer define next build
  - Difficult to have initial high-level design
    - Incremental testing is difficult
    - System testing is not affected
- Spiral
  - Combination of incremental and evolutionary
  - After each build assess benefits and risks
    - Use to decide go/no-go and direction
  - Difficult to have initial high-level design
    - Incremental testing is difficult
    - System testing is not affected
- Advantage of spiral models
  - Earlier synthesis and deliverables
  - More customer feedback

- Risk/benefit analysis is rigorous

Specification Based Life Cycle Models

When systems are not fully understood (by either the customer or the developer), functional decomposition is perilous at best. Barry Boehm jokes when he describes the customer who says "I don't know what I want, but I'll recognize it when I see it." The rapid prototyping life cycle deals with this by providing the "look and feel" of a system. Thus, in a sense, customers can recognize what they "see." In turn, this drastically reduces the pecification-to-customer feedback loop by producing very early synthesis. Rather than build a final system, a "quick and dirty" prototype is built and then used to elicit customer feedback. Depending on the feedback, more prototyping cycles may occur. Once the developer and the customer agree that a prototype represents the desired system, the developer goes ahead and builds to a correct specification. At this point, any of the waterfall spin-offs might also be used. The agile life cycles are the extreme of this pattern. Rapid prototyping has no new implications for integration testing; however, it has very interesting implications for system testing. Where are the requirements? Is the last prototype the specification? How are system test cases traced back to the prototype? One good answer to questions such as these is to use the prototyping cycles as information-gathering activities and then produce a requirements specification in a more traditional manner. Another possibility is to capture what the customer does with the prototypes, define these as scenarios that are important to the customer, and then use these as system test cases. These could be precursors to the user stories of the agile life cycles. The main contribution of rapid prototyping is that it brings the operational (or behavioral) viewpoint to the requirements specification phase. Usually, requirements specification techniques emphasize the structure of a system, not its behavior. This is unfortunate because most customers do not care about the structure, and they do care about the behavior. Executable specifications are an extension of the rapid prototyping concept. With this approach, the requirements are specified in an executable format (such as finite state machines, StateCharts, or Petri nets). The customer then executes the specification to observe the intended system behavior and provides feedback as in the rapid prototyping model. The executable models are, or can be, quite complex. This is an understatement for the full-blown version of StateCharts. Building an executable model requires expertise, and executing it requires an engine. Executable specification is best applied to event-driven systems, articularly when the events can arrive in different orders. David Harel, the creator of StateCharts, refers to such systems as "reactive" (Harel, 1988) because they react to external events. As with rapid prototyping, the purpose of an executable specification is to let the customer experience scenarios of intended behavior. Another similarity is that executable models might have to be revised on the basis of customer feedback. One side benefit is that a good engine for an executable model will support the capture of "interesting" system transactions, and it is often a nearly mechanical process to convert these into true system test cases. If this is done carefully, system testing can be traced directly back to the requirements. Once again, this life cycle has no implications for integration testing. One big difference is that the requirements specification document is explicit, as opposed to a prototype. More important, it is often a mechanical process to derive system test cases from an executable specification. Although more work is required to develop an executable specification, this is partially offset by the reduced effort to generate system test cases. Here is another important distinction: when system testing is based on an executable specification, we have an interesting form of structural testing at the system level. Finally, as we saw with rapid prototyping, the executable specification step can be combined with any of the iterative life cycle models.



9. A.

SCAFFOLDING

 Code developed to facilitate testing is called scaffolding, by analogy to the temporary structures erected around a building during construction or maintenance.
 Scaffoldings may include

→Test drivers (substituting for a main or calling population) →Test harness (substituting for parts of the deployment environment) →Stubs (substituting for functionally called or used by the software under test)
 The purpose of scaffolding is to provide controllability to execute test cases and observability to judge the outcome of test execution.
 Sometimes scaffolding is required to simply make module executable, but even in incremental development with immediate integration of each module, scaffolding for controllability and observability may be required because the external interfaces of the system may not provide sufficient control to drive the module under test through test cases, or sufficient observability of the effect.
 Example: consider an interactive program that is normally driven through a GUI. Assume that each night the person goes through a fully automate and unattended cycle of integration compilation, and test execution.
 It is necessary to perform some testing through the interactive interface, but it is neither necessary nor efficient to execute all test cases that way. Small driver programs, independent of GUI can drive each module through large test suites in a short time.

GENERIC VERSUS SPECIFIC SCAFFOLDING

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

9. B. What are Planning and Monitoring?
• Planning:
– Scheduling activities (what steps? in what order?)
– Allocating resources (who will do it?)
– Devising unambiguous milestones for monitoring
• Monitoring: Judging progress against the plan
– How are we doing?
• A good plan must have *visibility* :
– Ability to monitor each step, and to make objective judgments of progress

– Counter wishful thinking and denial

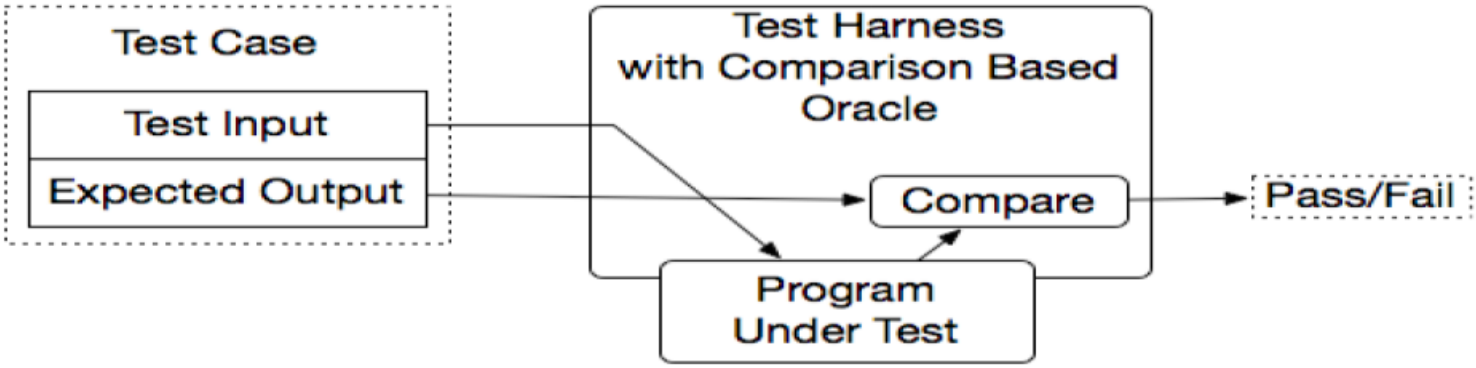
Quality process: Set of activities and responsibilities
– focused primarily on ensuring adequate dependability
– concerned with project schedule or with product usability

• A framework for
– selecting and arranging activities
– considering interactions and trade-offs
• Follows the overall software process in which it is embedded
– Example: waterfall software process —> "V model": unit testing starts with implementation and finishes before integration
– Example: XP and agile methods —> emphasis on unit testing and rapid iteration for acceptance testing by customers
Key principle of quality planning
– *the cost of detecting and repairing a fault increases as a function of time between committing an error and detecting the resultant faults*
• therefore ...
– an efficient quality plan includes matched sets of *intermediate* validation and verification activities that *detect most faults within a short time* of their introduction
• and ...
– V&V steps depend on the intermediate work products and on their *anticipated defects*

10. A.

TEST ORACLES

 In practice, the pass/fail criterion is usually imperfect.
 A test oracle may apply a pass/fail criterion that reflects only a part of the actual program specification, or is an approximation, and therefore passes some program executions it ought to fail
 Several partial test oracles may be more cost-effective than one that is more comprehensive
 A test oracle may also give false alarms, failing an execution that is ought to pass.
 False alarms in test execution are highly undesirable.
 The best oracle we can obtain is an oracle that detects deviations from expectation that may or may not be actual failure.



Two types

❖ **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.
- It is best suited for small number of hand generated test cases example: for handwritten Junit test cases.
- They are used mainly for small, simple test cases
- Expected outputs can also be produced for complex test cases and large test suites
- Capture-replay testing, a special case in which the predicted output or behavior is preserved from an earlier execution
- Often possible to judge output or behavior without predicting it

❖ **Partial oracle**

- Oracles that check results without references to predicted output are often partial, in the sense that they can detect some violations of the actual specification but not others.
- They check necessary but not sufficient conditions for correctness.

- A cheap partial oracle that can be used for a large number of test cases is often combined with a more expensive comparison-based oracle that can be used with a smaller set of test cases for which predicted output has been obtained
- Specifications are often incomplete
- Automatic derivations of test oracles are impossible

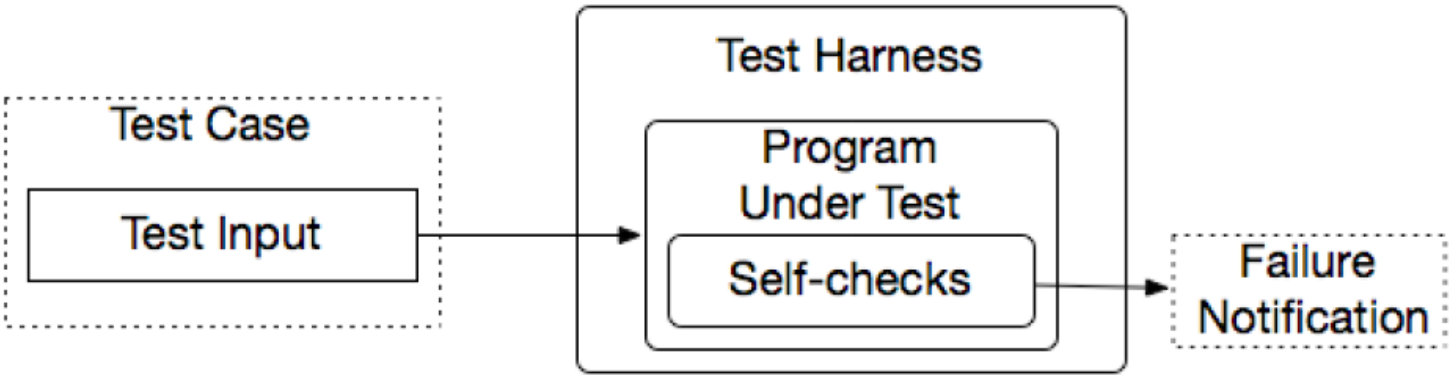
SELF-CHECKS AS ORACLES

 An oracle can also be written as self checks

-Often possible to judge correctness without predicting results.
 Typically these self checks are in the form of assertions, but designed to be checked during execution.
 It is generally considered good design practice to make assertions and self checks to be free of side effects on program state.
 Self checks in the form of assertions embedded in program code are useful primarily for checking module and subsystem-level specification rather than all program behaviour.
 Devising the program assertions that correspond in a natural way to specifications poses two main challenges:
Bridging the gap between concrete execution values and abstractions used in specification
Dealing in a reasonable way with quantification over collection of values

• Structural invariants are good candidates for self checks implemented as assertions
 They pertain directly to the concrete data structure implementation
 It is sometimes straight-forward to translate quantification in a specification statement into iteration in a program assertion
 A run time assertion system must manage ghost variables
 They must retain “before” values
 They must ensure that they have no side effects outside assertion checking
 Advantages:
-Usable with large, automatically generated test suites.
 Limits:

-often it is only a partial check. -recognizes many or most failures, but not all.



10. B. What is Mutation Testing?

Mutation testing is a structural testing technique, which uses the structure of the code to guide the testing process. On a very high level, it is the process of rewriting the source code in small ways in order to remove the redundancies in the source code

These ambiguities might cause failures in the software if not fixed and can easily pass through testing phase undetected.

Mutation Testing Benefits:

Following benefits are experienced, if mutation testing is adopted:

- It brings a whole new kind of errors to the developer's attention.
- It is the most powerful method to detect hidden defects, which might be impossible to identify using the conventional testing techniques.
- Tools such as Insure++ help us to find defects in the code using the state-of-the-art.
- Increased customer satisfaction index as the product would be less buggy.
- Debugging and Maintaining the product would be more easier than ever.

Mutation Testing Types:

- **Value Mutations:** An attempt to change the values to detect errors in the programs. We usually change one value to a much larger value or one value to a much smaller value. The most common strategy is to change the constants.
- **Decision Mutations:** The decisions/conditions are changed to check for the design errors. Typically, one changes the arithmetic operators to locate the defects and also we can consider mutating all relational operators and logical operators (AND, OR , NOT)
- **Statement Mutations:** Changes done to the statements by deleting or duplicating the line which might arise when a developer is copy pasting the code from somewhere else.