USN

13MCA444

Fourth Semester MCA Degree Examination, June/July 2017

Software Testing & Practices

Time: 3 hrs. Max. Marks:100

Note: *Answer any FIVE full questions.*

Important Note : 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.

1 a. Briefly explain about functional testing and structural testing. **(10 Marks)**

b. Explain about quality attributes. **(05 Marks)**

c. Differentiate between testing and debugging. **(05 Marks)**

2 a. Generate the BOR – constraint set and construct an abstract syntax tree of predicate $P_r = (a+b) < c \wedge !P \vee (r > s)$. Write an algorithm to generate a minimal BOR-constraint set from an abstract syntax tree of a predicate P_r. **(10 Marks)**

b. Explain the six basic principles of software testing. **(10 Marks)**

3 a. Brief out the program behavior to draw the venn diagram. **(10 Marks)**

b. State and explain the data flow diagram for the triangle problem. **(05 Marks)**

c. Describe about SATM screens with the problem statements. **(05 Marks)**

4 a. Explain boundary value analysis and generalizing boundary value analysis. **(10 Marks)**

b. Write equivalence class test cases for triangle problem. **(05 Marks)**

c. Define the decision table with an example and explain. **(05 Marks)**

5 a. Define the DD-path and write the DD-path for triangle program. **(10 Marks)**

b. Write the program-graph for the triangle program. **(05 Marks)**

c. Explain the data flow testing and slice – based testing with example. **(05 Marks)**

6 a. Differentiate between the traditional view of testing levels and alternative life cycle models. **(10 Marks)**

b. Explain about levels of testing. **(10 Marks)**

7 a. Explain about mutation analysis and fault based adequacy criteria. **(10 Marks)**

b. Brief about test oracles, self check as oracle. **(05 Marks)**

c. Describe about capture and replay. **(05 Marks)**

8 a. Justify with an example why document analysis is required. **(10 Marks)**

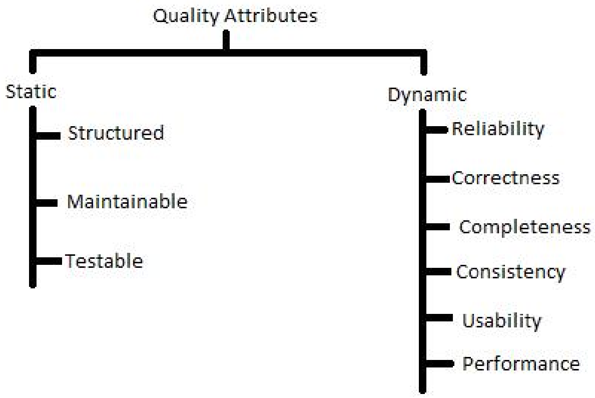
b. Briefly describe the test analysis document, design specification and report. **(10 Marks)**

* * * * *

1a. **Structural and Functional Technique**

- Both Structural and Functional Technique is used to ensure adequate testing
- Structural analysis basically test the uncover error occur during the coding of the program.
- Functional analysis basically test he uncover occur during implementing requirements and design specifications.
- Functional testing basically concern about the results but not the processing.
- Structural testing is basically concern both the results and also the process.
- Structural testing is used in all the phases where design, requirements and algorithm is discussed.
- The main objective of the Structural testing to ensure that the functionality is working fine and the product is technically good enough to implement in the real environment.
- Functional testing is sometimes called as black box testing, no need to know about the coding of the program.
- Structural testing is sometimes called as white box testing because knowledge of code is very much essential. We need to understand the code written by other users.

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."

c.

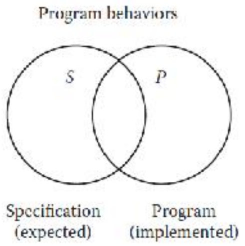
Testing	Debugging
1. Testing always starts with known conditions, uses predefined methods, and has predictable outcomes too.	1. Debugging starts from possibly un-known initial conditions and its end cannot be predicted, apart from statistically.
2. Testing can and should definitely be planned, designed, and scheduled.	2. The procedures for, and period of, debugging cannot be so constrained.
3. It proves a programmers failure.	3. It is the programmer’s vindication.
4. It is a demonstration of error or apparent correctness.	4. It is always treated as a deductive process.
5. Testing as executed should strive to be predictable, dull, constrained, rigid, and inhuman.	5. Debugging demands intuitive leaps, conjectures, experimentation, and some freedom also.
6. Much of the testing can be done without design knowledge.	6. Debugging is impossible without detailed design knowledge.
7. It can often be done by an outsider.	7. It must be done by an insider.
8. Much of test execution and design can be automated.	8. Automated debugging is still a dream for programmers.
9. Testing purpose is to find bug.	9. Debugging purpose is to find cause of bug.

2a.

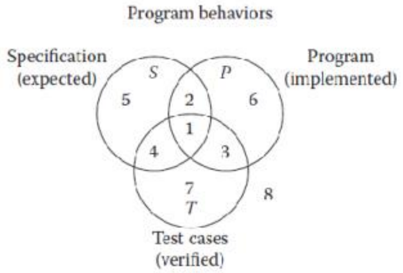
b. Basic principles of software testing -

- 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

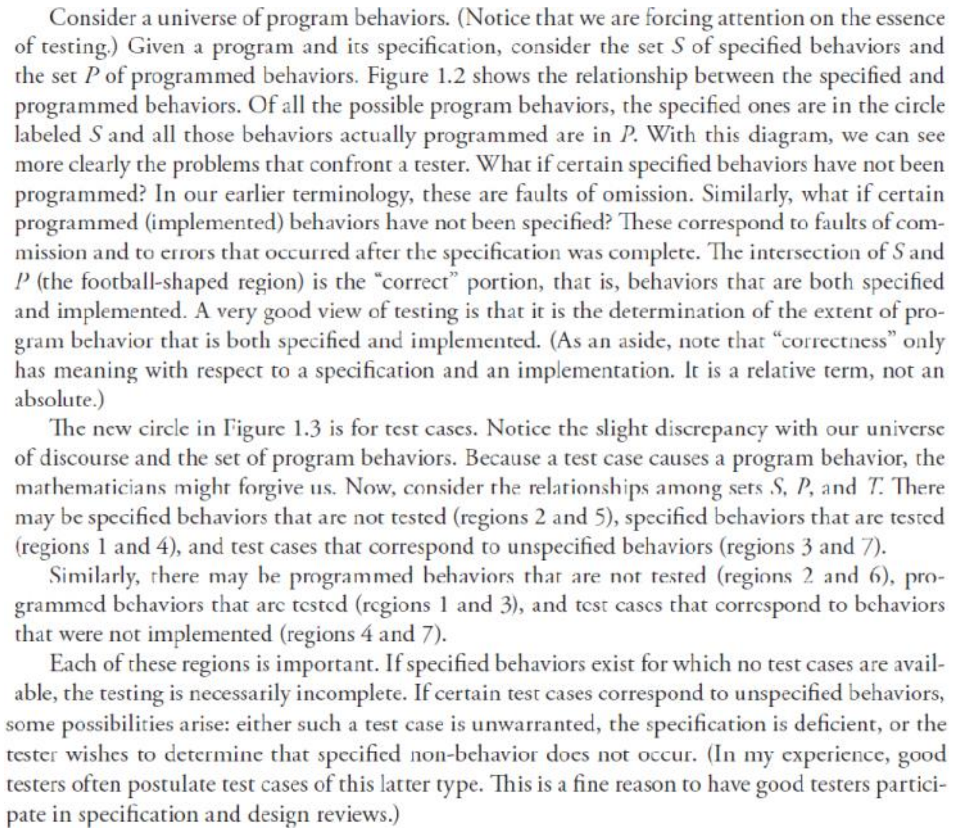
3a.



Specified and implemented program behaviors.



Specified, implemented, and tested behaviors.



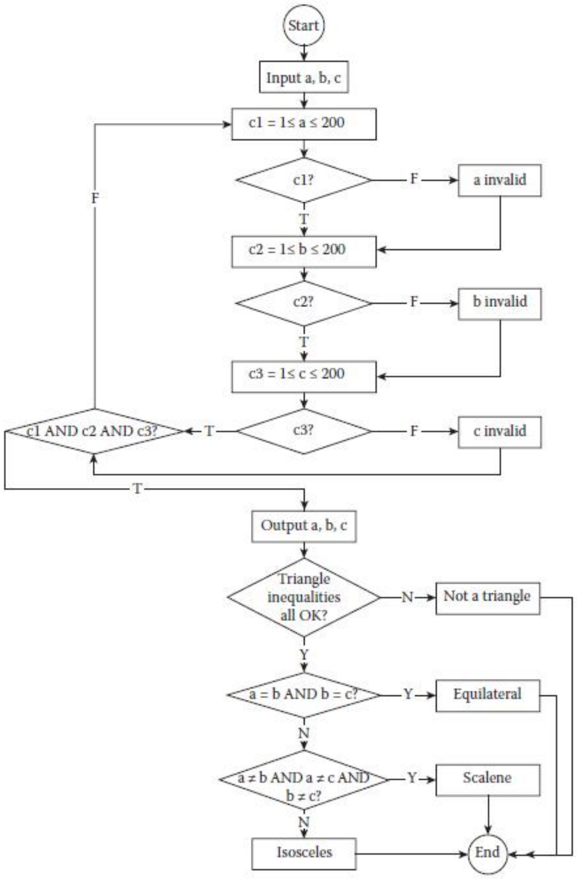
Consider a universe of program behaviors. (Notice that we are forcing attention on the essence of testing.) Given a program and its specification, consider the set *S* of specified behaviors and the set *P* of programmed behaviors. Figure 1.2 shows the relationship between the specified and programmed 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).

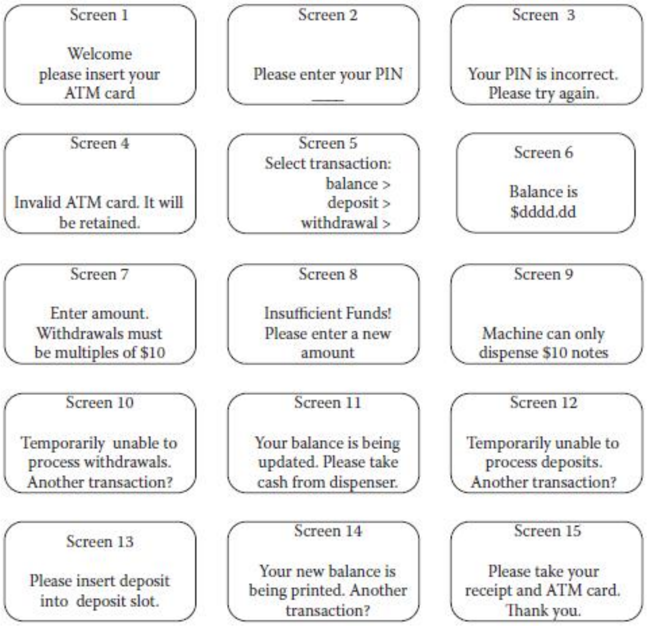
Each of these regions is important. If specified behaviors exist for which no test cases are available, the testing is necessarily incomplete. If certain test cases correspond to unspecified behaviors, some possibilities arise: either such a test case is unwarranted, the specification is deficient, or 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.)

b.



c. The SATM system communicates with bank customers via the 15 screens shown in following figure. 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. If a problem occurs with the deposit envelope slot, the system displays screen 12. Once the deposit amount has been entered, the system displays screen 13, accepts the deposit envelope, and processes the deposit. The system then displays screen 14. If a withdrawal is requested, the system checks the status (jammed or free) of the withdrawal chute in the terminal control file. If jammed, screen 10 is displayed; otherwise, screen 7 is displayed so the customer can enter the withdrawal amount. Once the withdrawal amount is entered, the system checks the terminal status file to see if it has enough currency to dispense. If it does not, screen 9 is displayed; otherwise, the withdrawal is processed. The system checks the customer balance (as described in the balance request transaction); if the funds in the account are insufficient, screen 8 is displayed. If the account balance is sufficient, screen 11 is displayed and the money is dispensed. The balance is printed on the transaction receipt as it is for a balance request transaction. After the cash has been removed, the system displays screen 14. When the "No" button is pressed in screens 10, 12, or 14, the system presents screen 15 and returns the customer's ATM card. Once the card is removed from the card slot, screen 1 is displayed. When the "Yes" button is pressed in screens 10, 12, or 14, the system presents screen 5 so the customer can select additional transactions.


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?

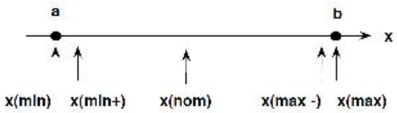
Screen 15

Please take your
receipt and ATM card.
Thank you.

4a. Boundary Value Analysis: Boundary testing is the process of testing between extreme ends or boundaries between partitions of the input values.

- So these extreme ends like Start- End, Lower- Upper, Maximum-Minimum, Just Inside-Just Outside values are called boundary values and the testing is called "boundary testing".
- The basic idea in boundary value testing is to select input variable values at their:

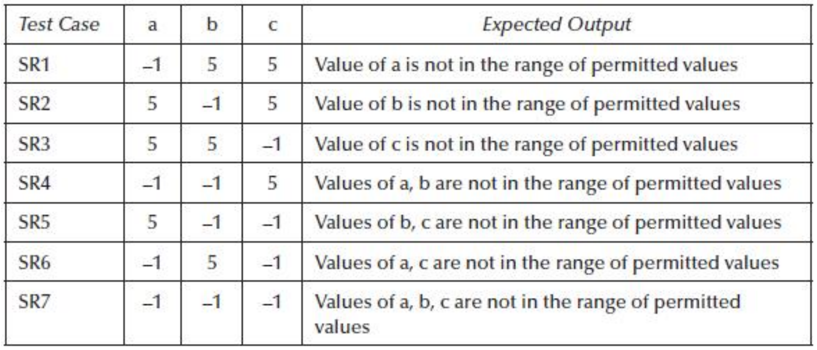
1. Minimum
2. Just above the minimum
3. A nominal value
4. Just below the maximum
5. Maximum



- In Boundary Testing, Equivalence Class Partitioning plays a good role
- Boundary Testing comes after the Equivalence Class Partitioning.

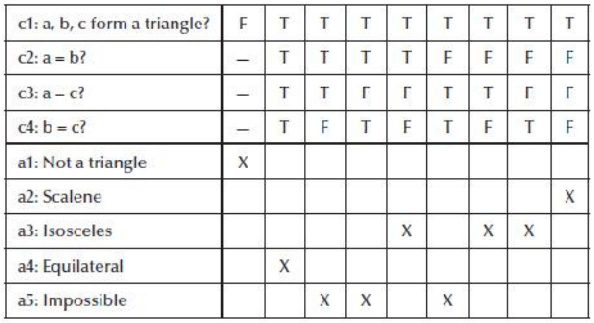
Generalizing boundary value analysis: There are two approaches to generalizing Boundary Value Analysis. We can do this by the number of variables or by the ranges these variables use. To generalise by the number of variables is relatively simple. This is the approach taken as shown by the general Boundary Value Analysis technique using the critical fault assumption. Generalizing by ranges depends on the type of the variables. For example in the NextDate example proposed by P.C. Jorgensen, we have variable for the year, month and day. Languages similar to the likes of FORTRAN would normally encode the month's variable so that January corresponded to 1 and February corresponded to 2 etc. Also it would be possible in some languages to declare an enumerated type {Jan, Feb, Mar,......, Dec}. Either way this type of declaration is relatively simple because the ranges have set values. When we do not have explicit bounds on these variable ranges then we have to create our own. These are known as artificial bounds and can be illustrated via the use of the Tri angle problem. The point raised by P.C. Jorgensen was that we can easily impose a lower bound on the length of an edge for the triangle as an edge with a negative length would be "silly". The problem occurs when trying to decide upon an upper bound for the length of each length. We could use a certain set integer, we could allow the program to use the highest possible integer (normally denoted as something to the effect of MaxInt). The arbitrary nature of this problem can lead to messy results or non concise test cases.

b. Equivalence Class Test Cases for Triangle Problem:



Test Case	a	b	c	*Expected Output*
SR1	–1	5	5	Value of a is not in the range of permitted values
SR2	5	–1	5	Value of b is not in the range of permitted values
SR3	5	5	–1	Value of c is not in the range of permitted values
SR4	–1	–1	5	Values of a, b are not in the range of permitted values
SR5	5	–1	–1	Values of b, c are not in the range of permitted values
SR6	–1	5	–1	Values of a, c are not in the range of permitted values
SR7	–1	–1	–1	Values of a, b, c are not in the range of permitted values

c. Decision Table: Decision tables have been used to represent and analyze complex logical relationships since the early 1960s. They are ideal for describing situations in which a number of combinations of actions are taken under varying sets of conditions. A decision table has four portions: the part to the left of the bold vertical line is the stub portion; to the right is the entry portion. The part above the bold horizontal line is the condition portion, and below is the action portion. Thus, we can refer to the condition stub, the condition entries, the action stub, and the action entries. A column in the entry portion is a rule. Rules indicate which actions, if any, are taken for the circumstances indicated in the condition portion of the rule. Following example is the decision table of the triangle problem



c1: a, b, c form a triangle?	F	T	T	T	T	T	T	T	T
c2: a = b?	—	T	T	T	T	F	F	F	F
c3: a – c?	—	T	T	F	F	T	T	F	F
c4: b = c?	—	T	F	T	F	T	F	T	F
a1: Not a triangle	X								
a2: Scalene									X
a3: Isosceles					X		X	X	
a4: Equilateral		X							
a5: Impossible			X	X		X			

5a. DD Path - A *DD-path* is a sequence of nodes in a program graph such that
Case 1: It consists of a single node with indeg = 0.
Case 2: It consists of a single node with outdeg = 0.
Case 3: It consists of a single node with indeg ≥ 2 or outdeg ≥ 2.
Case 4: It consists of a single node with indeg = 1 and outdeg = 1.
Case 5: It is a maximal chain of length ≥ 1.

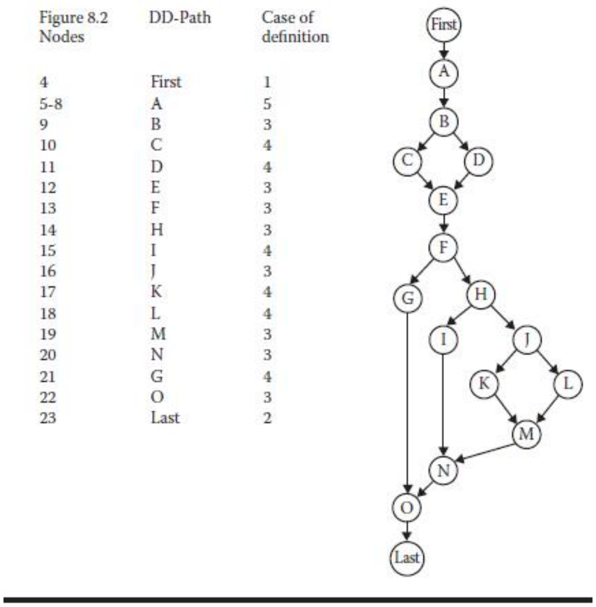
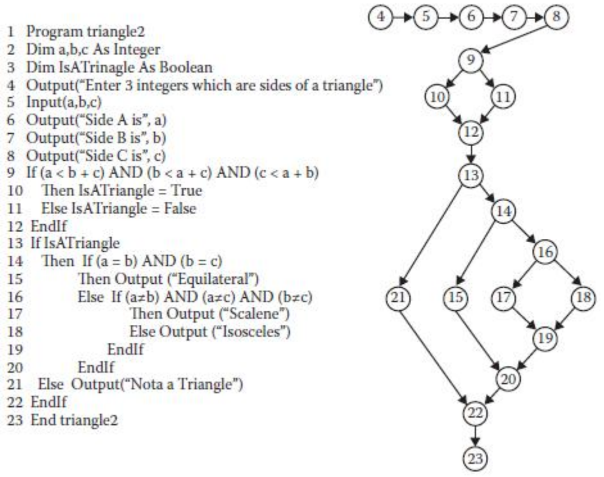


Figure 8.2 Nodes	DD-Path	Case of definition
4	First	1
5-8	A	5
9	B	3
10	C	4
11	D	4
12	E	3
13	F	3
14	H	3
15	I	4
16	J	3
17	K	4
18	L	4
19	M	3
20	N	3
21	G	4
22	O	3
23	Last	2

DD-path graph for triangle program.

b.



```
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.

c. Data Flow Testing: Data flow testing is a family of test strategies based on selecting paths through the program's control flow in order to explore sequences of events related to the status of variables or data objects. Dataflow Testing focuses on the points at which variables receive values and the points at which these values are used.

Data Flow testing helps us to pinpoint any of the following issues:

- A variable that is declared but never used within the program.
- A variable that is used but never declared.
- A variable that is defined multiple times before it is used.
- Deallocating a variable before it is used.

Slice-Based Testing Definitions

- Given a program P, and a program graph G(P) in which statements and statement fragments are numbered, and a set V of variables in P, the ***slice on the variable set V at statement fragment n***, written S(V,n), is the set node numbers of all statement fragments in P prior to n that contribute to the values of variables in V at statement fragment n
- The idea of slices is to separate a program into components that have some useful meaning

- We will include CONST declarations in slices
- Five forms of usage nodes
  - P-use (used in a predicate (decision))
  - C-use (used in computation)

    - O-use (used for output, e.g. writeln())
    - L-use (used for location, e.g. pointers)
    - I-use (iteration, e.g. internal counters)
- Two forms of definition nodes
    - I-def (defined by input, e.g. readln())
    - A-def (defined by assignment)
- For now, we presume that the slice S(V,n) is a slice on one variable, that is, the set V consists of a single variable, v

- If statement fragment n (in S(V,n)) is a defining node for v, then n is included in the slice
- If statement fragment n (in S(V,n)) is a usage node for v, then n is not included in the slice
- P-uses and C-uses of other variables are included to the extent that their execution affects the value of the variable v
- O-use, L-use, and I-use nodes are excluded from slices
- Consider making slices compliable

Slice-Based Testing Examples

- Find the following program slices
- S(commission,48)
- S(commission,40)
- S(commission,39)
- S(commission,38)
- S(sales,35)
- S(num_locks,34)
- S(num_stocks,34)
- S(num_barrels,34)

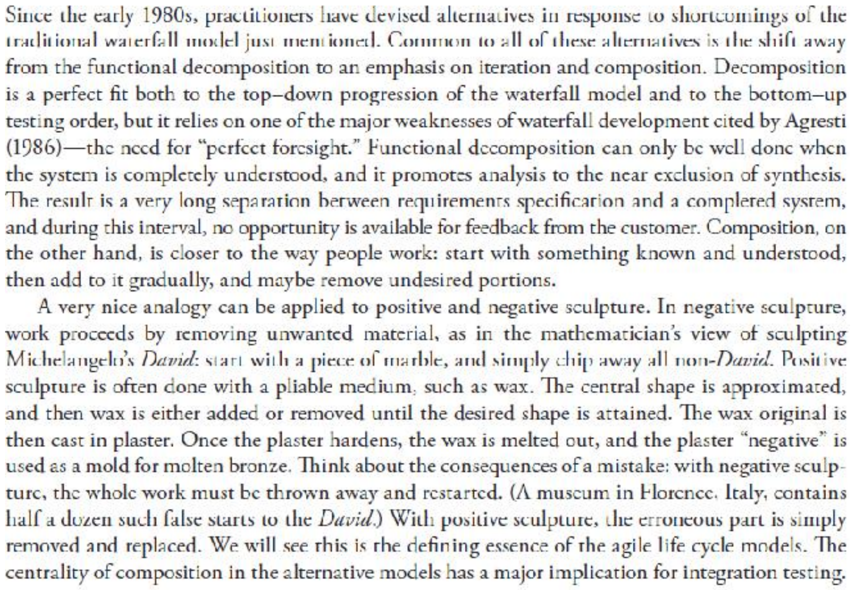
- S(commission,48)
    - {1-5,8-11,13,14,19-30,36,47,48,53}
- S(commission,40), S(commission,39), S(commission,38)
    - {Ø}
- S(sales,35)
    - {Ø}

- S(num_locks,34)
    - {1,8,9,10,13,14,19, 22,23,24,26,29,30, 53}

- S(num_stocks,34)
    - {1,8,9,10,13,14,20, 22-25,27,29,30,53}

- S(num_barrels,34)
    - {1,8,9,10,13,14,21-25,28,29,30,53}

6a.



Since the early 1980s, practitioners have devised alternatives in response to shortcomings of the traditional waterfall model just mentioned. Common to all of these alternatives is the shift away from the functional decomposition to an emphasis on iteration and composition. Decomposition is a perfect fit both to the top–down progression of the waterfall model and to the bottom–up testing order, but it relies on one of the major weaknesses of waterfall development cited by Agresti (1986)—the need for "perfect foresight." Functional decomposition can only be well done when the system is completely understood, and it promotes analysis to the near exclusion of synthesis. The result is a very long separation between requirements specification and a completed system, and during this interval, no opportunity is available for feedback from the customer. Composition, on the other hand, is closer to the way people work: start with something known and understood, then add to it gradually, and maybe remove undesired portions.

A very nice analogy can be applied to positive and negative sculpture. In negative sculpture, work proceeds by removing unwanted material, as in the mathematician's view of sculpting Michelangelo's *David*: start with a piece of marble, and simply chip away all non-*David*. Positive sculpture is often done with a pliable medium, such as wax. The central shape is approximated, and then wax is either added or removed until the desired shape is attained. The wax original is then cast in plaster. Once the plaster hardens, the wax is melted out, and the plaster "negative" is used as a mold for molten bronze. Think about the consequences of a mistake: with negative sculpture, the whole work must be thrown away and restarted. (A museum in Florence, Italy, contains half a dozen such false starts to the *David*.) With positive sculpture, the erroneous part is simply removed and replaced. We will see this is the defining essence of the agile life cycle models. The centrality of composition in the alternative models has a major implication for integration testing.

6b.There are different levels during the process of testing. In this chapter, a brief description is provided about these levels.

Levels of testing include different methodologies that can be used while conducting software testing. The main levels of software testing are:

- Functional Testing
- Non-functional Testing

Functional Testing

This is a type of black-box testing that is based on the specifications of the software that is to be tested. The application is tested by providing input and then the results are examined that need to conform to the functionality it was intended for. Functional testing of a software is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.

There are five steps that are involved while testing an application for functionality.

Steps	**Description**
I	The determination of the functionality that the intended application is meant to perform.
II	The creation of test data based on the specifications of the application.
III	The output based on the test data and the specifications of the application.
IV	The writing of test scenarios and the execution of test cases.
V	The comparison of actual and expected results based on the executed test cases.

An effective testing practice will see the above steps applied to the testing policies of every organization and hence it will make sure that the organization maintains the strictest of standards when it comes to software quality.

Unit Testing

This type of testing is performed by developers before the setup is handed over to the testing team to formally execute the test cases. Unit testing is performed by the respective developers on the individual units of source code assigned areas. The developers use test data that is different from the test data of the quality assurance team.

The goal of unit testing is to isolate each part of the program and show that individual parts are correct in terms of requirements and functionality.

Limitations of Unit Testing

Testing cannot catch each and every bug in an application. It is impossible to evaluate every execution path in every software application. The same is the case with unit testing.

There is a limit to the number of scenarios and test data that a developer can use to verify a source code. After having exhausted all the options, there is no choice but to stop unit testing and merge the code segment with other units.

Integration Testing

Integration testing is defined as the testing of combined parts of an application to determine if they function correctly. Integration testing can be done in two ways: Bottom-up integration testing and Top-down integration testing.

S.N.	**Integration Testing Method**
1	Bottom-up integration This testing begins with unit testing, followed by tests of progressively higher-level combinations of units called modules or builds.
2	Top-down integration In this testing, the highest-level modules are tested first and progressively, lower-level modules are tested thereafter.

In a comprehensive software development environment, bottom-up testing is usually done first, followed by top-down testing. The process concludes with multiple tests of the complete application, preferably in scenarios designed to mimic actual situations.

System Testing

System testing tests the system as a whole. Once all the components are integrated, the application as a whole is tested rigorously to see that it meets the specified Quality Standards. This type of testing is performed by a specialized testing team.

System testing is important because of the following reasons:

- System testing is the first step in the Software Development Life Cycle, where the application is tested as a whole.
- The application is tested thoroughly to verify that it meets the functional and technical specifications.
- The application is tested in an environment that is very close to the production environment where the application will be deployed.
- System testing enables us to test, verify, and validate both the business requirements as well as the application architecture.

Regression Testing

Whenever a change in a software application is made, it is quite possible that other areas within the application have been affected by this change. Regression testing is performed to verify that a fixed bug hasn't resulted in another functionality or business rule violation. The intent of regression testing is to ensure that a change, such as a bug fix should not result in another fault being uncovered in the application.

Regression testing is important because of the following reasons:

- Minimize the gaps in testing when an application with changes made has to be tested.
- Testing the new changes to verify that the changes made did not affect any other area of the application.
- Mitigates risks when regression testing is performed on the application.
- Test coverage is increased without compromising timelines.
- Increase speed to market the product.

Acceptance Testing

This is arguably the most important type of testing, as it is conducted by the Quality Assurance Team who will gauge whether the application meets the intended specifications and satisfies the client’s requirement. The QA team will have a set of pre-written scenarios and test cases that will be used to test the application.

More ideas will be shared about the application and more tests can be performed on it to gauge its accuracy and the reasons why the project was initiated. Acceptance tests are not only intended to point out simple spelling mistakes, cosmetic errors, or interface gaps, but also to point out any bugs in the application that will result in system crashes or major errors in the application.

By performing acceptance tests on an application, the testing team will deduce how the application will perform in production. There are also legal and contractual requirements for acceptance of the system.

Alpha Testing

This test is the first stage of testing and will be performed amongst the teams (developer and QA teams). Unit testing, integration testing and system testing when combined together is known as alpha testing. During this phase, the following aspects will be tested in the application:

- Spelling Mistakes
- Broken Links
- Cloudy Directions
- The Application will be tested on machines with the lowest specification to test loading times and any latency problems.

Beta Testing

This test is performed after alpha testing has been successfully performed. In beta testing, a sample of the intended audience tests the application. Beta testing is also known as **pre-release testing**. Beta test versions of software are ideally distributed to a wide audience on the Web, partly to give the program a "real-world" test and partly to provide a preview of the next release. In this phase, the audience will be testing the following:

- Users will install, run the application and send their feedback to the project team.
- Typographical errors, confusing application flow, and even crashes.
- Getting the feedback, the project team can fix the problems before releasing the software to the actual users.
- The more issues you fix that solve real user problems, the higher the quality of your application will be.
- Having a higher-quality application when you release it to the general public will increase customer satisfaction.

Non-Functional Testing

This section is based upon testing an application from its non-functional attributes. Non-functional testing involves testing a software from the requirements which are nonfunctional in nature but important such as performance, security, user interface, etc.

Some of the important and commonly used non-functional testing types are discussed below.

Performance Testing

It is mostly used to identify any bottlenecks or performance issues rather than finding bugs in a software. There are different causes that contribute in lowering the performance of a software:

- Network delay
- Client-side processing
- Database transaction processing
- Load balancing between servers
- Data rendering

Performance testing is considered as one of the important and mandatory testing type in terms of the following aspects:

- Speed (i.e. Response Time, data rendering and accessing)
- Capacity
- Stability
- Scalability

Performance testing can be either qualitative or quantitative and can be divided into different sub-types such as **Load testing** and **Stress testing**.

Load Testing

It is a process of testing the behavior of a software by applying maximum load in terms of software accessing and manipulating large input data. It can be done at both normal and peak load conditions. This type of testing identifies the maximum capacity of software and its behavior at peak time.

Most of the time, load testing is performed with the help of automated tools such as Load Runner, AppLoader, IBM Rational Performance Tester, Apache JMeter, Silk Performer, Visual Studio Load Test, etc.

Virtual users (VUsers) are defined in the automated testing tool and the script is executed to verify the load testing for the software. The number of users can be increased or decreased concurrently or incrementally based upon the requirements.

Stress Testing

Stress testing includes testing the behavior of a software under abnormal conditions. For example, it may include taking away some resources or applying a load beyond the actual load limit.

The aim of stress testing is to test the software by applying the load to the system and taking over the resources used by the software to identify the breaking point. This testing can be performed by testing different scenarios such as:

- Shutdown or restart of network ports randomly
- Turning the database on or off

- Running different processes that consume resources such as CPU, memory, server, etc.

Usability Testing

Usability testing is a black-box technique and is used to identify any error(s) and improvements in the software by observing the users through their usage and operation.

According to Nielsen, usability can be defined in terms of five factors, i.e. efficiency of use, learn-ability, memory-ability, errors/safety, and satisfaction. According to him, the usability of a product will be good and the system is usable if it possesses the above factors.

Nigel Bevan and Macleod considered that usability is the quality requirement that can be measured as the outcome of interactions with a computer system. This requirement can be fulfilled and the end-user will be satisfied if the intended goals are achieved effectively with the use of proper resources.

Molich in 2000 stated that a user-friendly system should fulfill the following five goals, i.e., easy to Learn, easy to remember, efficient to use, satisfactory to use, and easy to understand.

In addition to the different definitions of usability, there are some standards and quality models and methods that define usability in the form of attributes and sub-attributes such as ISO-9126, ISO-9241-11, ISO-13407, and IEEE std.610.12, etc.

UI vs Usability Testing

UI testing involves testing the Graphical User Interface of the Software. UI testing ensures that the GUI functions according to the requirements and tested in terms of color, alignment, size, and other properties.

On the other hand, usability testing ensures a good and user-friendly GUI that can be easily handled. UI testing can be considered as a sub-part of usability testing.

Security Testing

Security testing involves testing a software in order to identify any flaws and gaps from security and vulnerability point of view. Listed below are the main aspects that security testing should ensure:

- Confidentiality
- Integrity
- Authentication
- Availability
- Authorization
- Non-repudiation
- Software is secure against known and unknown vulnerabilities
- Software data is secure
- Software is according to all security regulations
- Input checking and validation
- SQL insertion attacks
- Injection flaws
- Session management issues
- Cross-site scripting attacks
- Buffer overflows vulnerabilities
- Directory traversal attacks

Portability Testing

Portability testing includes testing software with the aim to ensure its reusability and that it can be moved from another software as well. Following are the strategies that can be used for portability testing:

- Transferring installed software from one computer to another.
- Building executable (.exe) to run the software on different platforms.

Portability testing can be considered as one of the sub-parts of system testing, as this testing type includes overall testing of a software with respect to its usage over different environments. Computer hardware, operating systems, and browsers are the major focus of portability testing. Some of the pre-conditions for portability testing are as follows:

- Software should be designed and coded, keeping in mind the portability requirements.
- Unit testing has been performed on the associated components.
- Integration testing has been performed.
- Test environment has been established.

7a. **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. Each mutated version is called a *mutant* and tests detect and reject mutants by causing the behavior of the original version to differ from the mutant. This is called *killing* the mutant. Test suites are measured by the percentage of mutants that they kill. New tests can be designed to kill additional mutants. Mutants are based on well-defined *mutation operators* that either mimic typical programming errors (such as using the wrong operator or variable name) or force the creation of valuable tests (such as dividing each expression by zero). The purpose is to help the tester develop effective tests or locate weaknesses in the test data used for the program or in sections of the code that are seldom or never accessed during execution. Mutation testing is a form of white-box testing.

Fault-based testing:

- Adequacy of test set = Ability to detect faults
- Methods:
  - Error seeding
  - Mutation testing
    - Program mutation testing
    - SPEC-mutation testing

b. Test Oracle: In computing, software testers and software engineers can use an **oracle** as a mechanism for determining whether a **test** has passed or failed. The use of oracles involves comparing the output(s) of the system under **test**, for a given **test**-case input, to the output(s) that the **oracle** determines that product should have.

Self-Checking Code as Oracle
• An oracle can also be written as *self-checks*
– Often possible to judge correctness without predicting results
• Advantages and limits: Usable with large, automatically generated test suites, but often only a *partial* check
– e.g., structural invariants of data structures

c. Capture and Replay
• Sometimes there is no alternative to human input and observation
– Even if we separate testing program functionality from GUI, some testing of the GUI is required
• We can at least cut *repetition* of human testing
• *Capture* a manually run test case, *replay* it automatically
– with a comparison-based test oracle: behavior same as previously accepted behavior
• reusable only until a program change invalidates it
• lifetime depends on abstraction level of input and output

8. a. Document analysis is a form of qualitative research in which documents are interpreted by the researcher to give voice and meaning around an assessment topic (Bowen, 2009). Analyzing documents incorporates coding content into themes similar to how focus group or interview transcripts are analyzed (Bowen,2009). A rubric can also be used to grade or score document. There are three primary types of documents (O'Leary, 2014):

- Public Records: The official, ongoing records of an organization's activities. Examples include student transcripts, mission statements, annual reports, policy manuals, student handbooks, strategic plans, and syllabi.
- Personal Documents: First-person accounts of an individual's actions, experiences, and beliefs. Examples include calendars, e-mails, scrapbooks, blogs, Facebook posts, duty logs, incident reports, reflections/journals, and newspapers.
- Physical Evidence: Physical objects found within the study setting (often called artifacts). Examples include flyers, posters, agendas, handbooks, and training materials.

Before actual document analysis takes place, the researcher must go through a detailed planning process in order to ensure reliable results. O'Leary outlines an 8-step planning process that should take place not just in document analysis, but all textual analysis (2014):

1. Create a list of texts to explore (e.g., population, samples, respondents, participants).
2. Consider how texts will be accessed with attention to linguistic or cultural barriers.
3. Acknowledge and address biases.
4. Develop appropriate skills for research.
5. Consider strategies for ensuring credibility.
6. Know the data one is searching for.
7. Consider ethical issues (e.g., confidential documents).
8. Have a backup plan.

A researcher can use a huge plethora of texts for research, although by far the most common is likely to be the use of written documents (O'Leary, 2014). There is the question of how many documents the researcher should gather. Bowen suggests that a wide array of documents is better, although the question should be more about quality of the document rather than quantity (Bowen, 2009). O'Leary also introduces two major issues to consider when beginning document analysis. The first is the issue of bias, both in the author or creator of the document, and the researcher as well (2014). The researcher must consider the subjectivity of the author and also the personal biases he or she may be bringing to the research. Bowen adds that the researcher must evaluate the original purpose of the document, such as the target audience (2009). He or she should also consider whether the author was a firsthand witness or used secondhand sources. Also important is determining whether the document was solicited, edited, and/or anonymous (Bowen, 2009). O'Leary's second major issue is the "unwitting" evidence, or latent content, of the document. Latent content refers to the style, tone, agenda, facts or opinions that exist in the document. This is a key first step that the researcher must keep in mind (O'Leary, 2014). Bowen adds that documents should be assessed for their completeness; in other words, how selective or comprehensive their data is (2009). Also of paramount importance when evaluating documents is not to consider the data as "necessarily precise, accurate, or complete recordings of events that have occurred" (Bowen, 2009, p. 33). These issues are summed up in another eight-step process offered by O'Leary (2014):

1. Gather relevant texts.
2. Develop an organization and management scheme.
3. Make copies of the originals for annotation.
4. Asses authenticity of documents.
5. Explore document's agenda, biases.
6. Explore background information (e.g., tone, style, purpose).
7. Ask questions about document (e.g., Who produced it? Why? When? Type of data?).
8. Explore content.

Step eight refers to the process of exploring the "witting" evidence, or the actual content of the documents, and O'Leary gives two major techniques for accomplishing this (2014). One is the interview technique. In this case, the researcher treats the document like a respondent or informant that provides the researcher with relevant information (O'Leary, 2014). The researcher "asks" questions then highlights the answer within the text. The other technique is noting occurrences, or content analysis, where the researcher quantifies the use of particular words, phrases and concepts (O'Leary, 2014). Essentially, the researcher determines what is being searched for, then documents and organizes the frequency and amount of occurrences within the document. The information is then organized into what is "related to central questions of the research" (Bowen, 2009, p. 32). Bowen notes that some experts object to this kind of analysis, saying that it obscures the interpretive process in the case of interview transcriptions (Bowen, 2009). However, Bowen reminds us that documents include a wide variety of types, and content analysis can be very useful for painting a broad, overall picture (2009). According to Bowen (2009), content analysis, then, is used as a "first-pass document review" (p. 32) that can provide the researcher a means of identifying meaningful and relevant passages.

b. TEST DESIGN SPECIFICATION

Test processes determine whether the development products of a given activity conform to the requirements of that activity and whether the system and/or software satisfies its intended use and user needs. The scope of testing encompasses software-based systems, computer software, hardware, and their interfaces.

829-2008 – IEEE Standard for Software and System Test Documentation, is an IEEE standard that specifies the form of a set of documents for use in eight defined stages of software testing, each stage potentially producing its own separate type of document.

Software quality assurance test documentation includes:

Test Design Specification
The test design is the first stage in developing the tests for software testing projects. It records what needs to be tested, and is derived from the documents that come into the testing stage, such as requirements and designs. It records which features of a test item are to be tested, and how a successful test of these features would be recognized.

The test design does not record the values to be entered for a test, but describes the requirements for defining those values.

This document is very valuable, but is often missing on many projects. The reason is that people start writing test cases before they have decided what they are going to test.

IEEE 829 Test Design Specification Template

Test Design Specification Identifier

• Specify the unique identifier assigned to this test procedure. Supply a reference to the associated test procedure specification. The naming convention should follow the same general rules as the software it is related, for coordinating software versions within configuration management.
• Unique “short” name for the case
• Version date and version number of the case
• Version Author and contact information
• Revision history

Features to be Tested

The set of test objectives covered by this test design specification. It is the overall purpose of this document to group related test items together.
• Features
• Attributes and Characteristics
• Groupings of features
• Level of testing appropriate to test item if this test design specification covers more that one level of testing
• Reference to the original documentation where this test objective (feature) was obtained.

Approach Refinements

Add the necessary level of detail and refinement based on the original approach defined in the test plan associated with this test design specification.
• Selection of specific test techniques
• Reasons for technique selection
• Method(s) for results analysis
• Tools etc.
• Relationship of the test items/features to the levels of testing
• Summarize any common information that may relate to multiple test cases or procedures. Centralizing common information reduces document size, and simplifies maintenance.
• Shared Environment
• Common setup/recovery
• Case dependencies

Test Identification

• Identification of each test case with a short description of the case, it’s test level or any other appropriate information required to describe the test relationship.
• Identification of each test procedure with a short description of the case, it’s test level or any other appropriate information required to describe the test relationship.

Feature Pass/Fail Criteria

Describe the criteria for assessing the feature or set of features and whether the test(s) were successful of not.