VTU-Exam-Feb-2021-Software Testing

Question Paper Answers

1. What is Software Quality? Explain the same with respect to quality attributes

1 a. SOFTWARE QUALITY

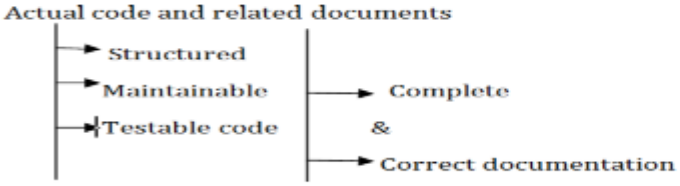
- Software quality is a multidimensional quantity and is measurable.

Quality Attributes

- These can be divided to static and dynamic quality attributes.

Static quality attributes

- It refers to the actual code and related documents.



Example: A poorly documented piece of code will be harder to understand and hence difficult to modify. A poorly structured code might be harder to modify and difficult to test.

Dynamic quality Attributes:

- Reliability
- Correctness

- Completeness
- Consistency
- Usability
- performance

Reliability:

- It refers to the probability of failure free operation.

Correctness:

- Refers to the correct operation and is always with reference to some artefact.
- For a Tester, correctness is w.r.t to the requirements
- For a user correctness is w.r.t the user manual

Completeness:

- Refers to the availability of all the features listed in the requirements or in the user manual.
- An incomplete software is one that does not fuly implement all features required.

Consistency:

- Refers to adherence to a common set of conventions and assumptions.
- Ex: All buttons in the user interface might follow a common-color coding convention.

Usability:

- Refer to 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.
- Usability testing is a testing done by its potential users.
- The development organization invites a selected set of potential users and asks them to test the product.
- Users in turn test for ease of use, functionality as expected, performance, safety and security.
- Users thus serve as an important source of tests that developers or testers within the organization might not have conceived.
- Usability testing is sometimes referred to as user-centric testing.

Performance:

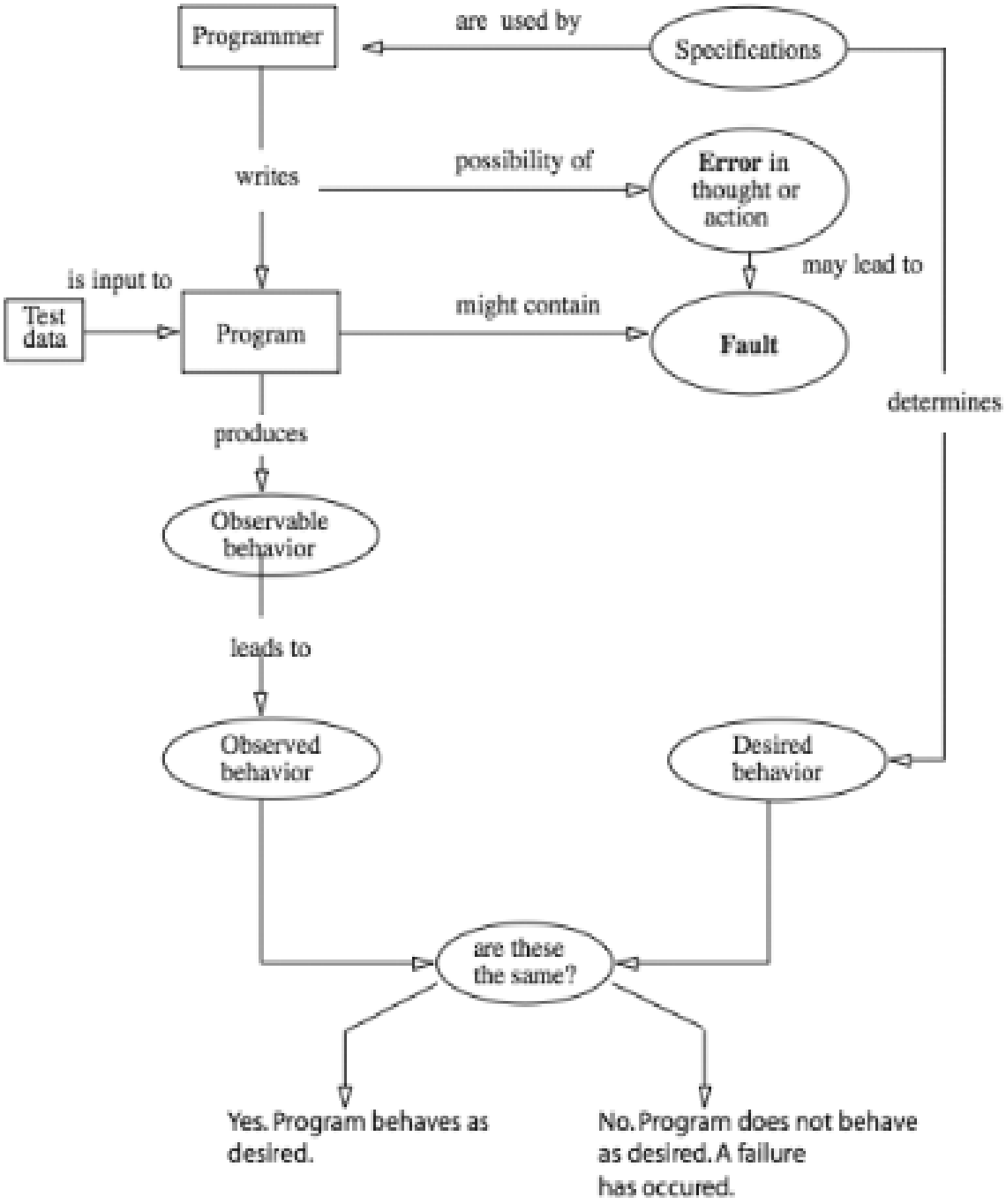
- Refers to the time the application takes to perform a requested task. Performance is considered as a non-functional requirement.

Reliability:

- (Software reliability is the probability of failure free operation of software over a given time interval & under given conditions.)
- Software reliability can vary from one operational profile to another. An implication is that one might say “this program is lousy” while another might sing praises for the same program.
- Software reliability is the probability of failure free operation of software in its intended environments.

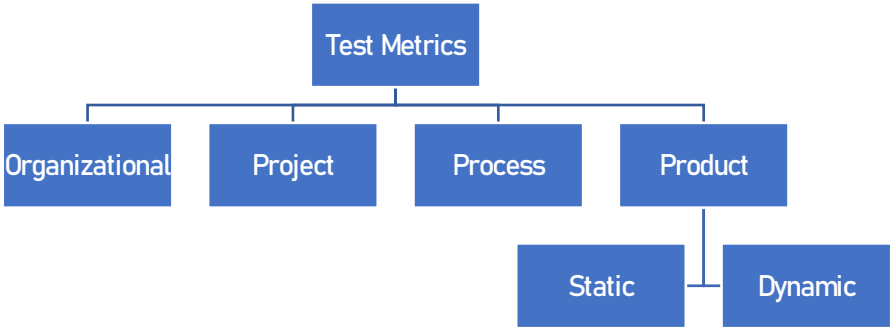
1b. Explain Errors , Faults, and Failures in the process of programming and testing with the diagram.

- An error occurs in the process of writing a program.
- A fault is the manifestation of one or more errors
- A failure occurs when a faulty piece of code is executed leading to an incorrect state that propagates to the program's output.



2a. Discuss different types of testing Matrices.

The term metric refers to a standard of measurement. In software testing there exist a variety of metrics



Organization

Establishes test processes

Used in projects

To test products

Organization Metrics: Metrics at the level of an organization are useful in overall project planning and management

Project Metrics: Project Metrics relate to a specific project

Process Metrics: Every project uses some test process.
The goal of a process metric is to assess the goodness of the process.

Product Metrics: Product metrics relates to a specific product such as a compiler for a programming language.

Static: Computed without having to execute the product

Dynamic metrics: Require code execution

2 d. Explain different steps of testing and debugging.

1. Identify the Error: A bad identification of an error can lead to wasted developing time. It is usual that production errors reported by users are hard to interpret and sometimes the information we receive is misleading. It is import to identify the actual error.

2. Find the Error Location: After identifying the error correctly, you need to go through the code to find the exact spot where the error is located. In this stage, you need to focus on finding the error instead of understanding it.

3. Analyze the Error: In the third step, you need to use a bottom-up approach from the error location and analyze the code. This helps you in understanding the error. Analyzing a bug has two main goals, such as checking around the error for other errors to be found, and to make sure about the risks of entering any collateral damage in the fix.

4. Prove the Analysis: Once you are done analyzing the original bug, you need to find a few more errors that may appear on the application. This step is about writing automated tests for these areas with the help of a test framework.

5. Cover Lateral Damage: In this stage, you need to create or gather all the unit tests for the code where you are going to make changes. Now, if you run these unit tests, they all should pass.

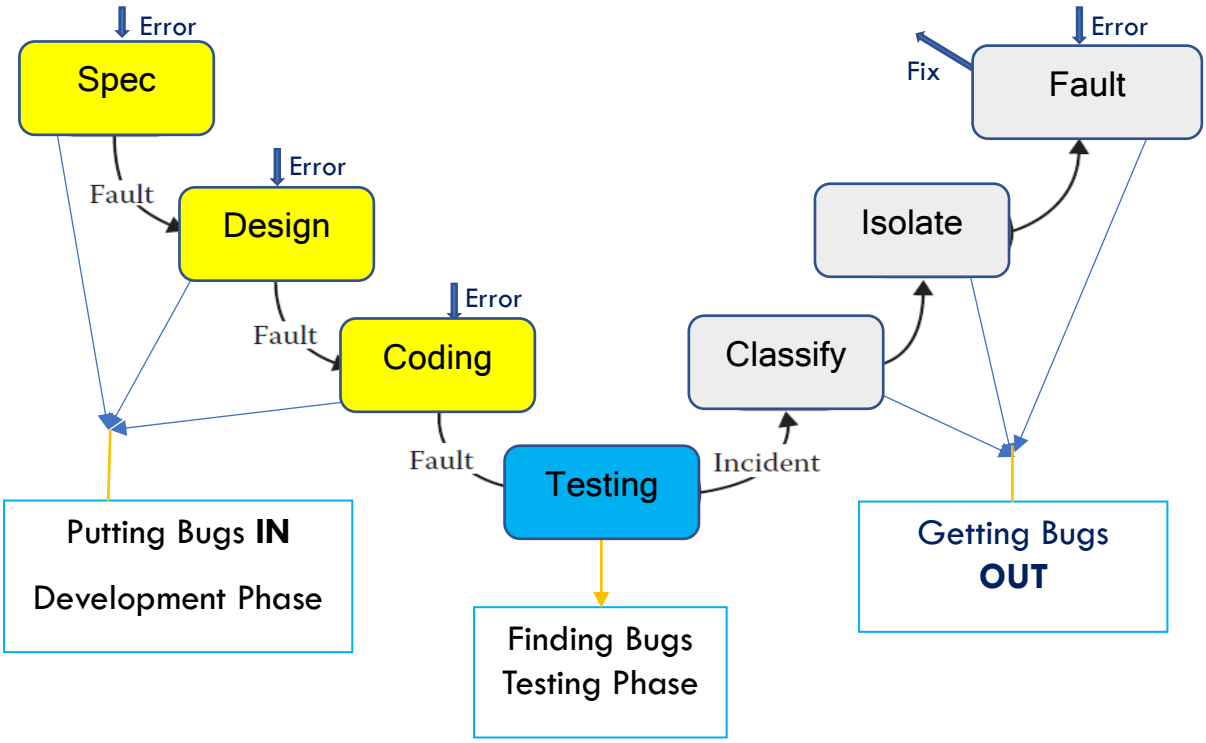
Debugging Strategies

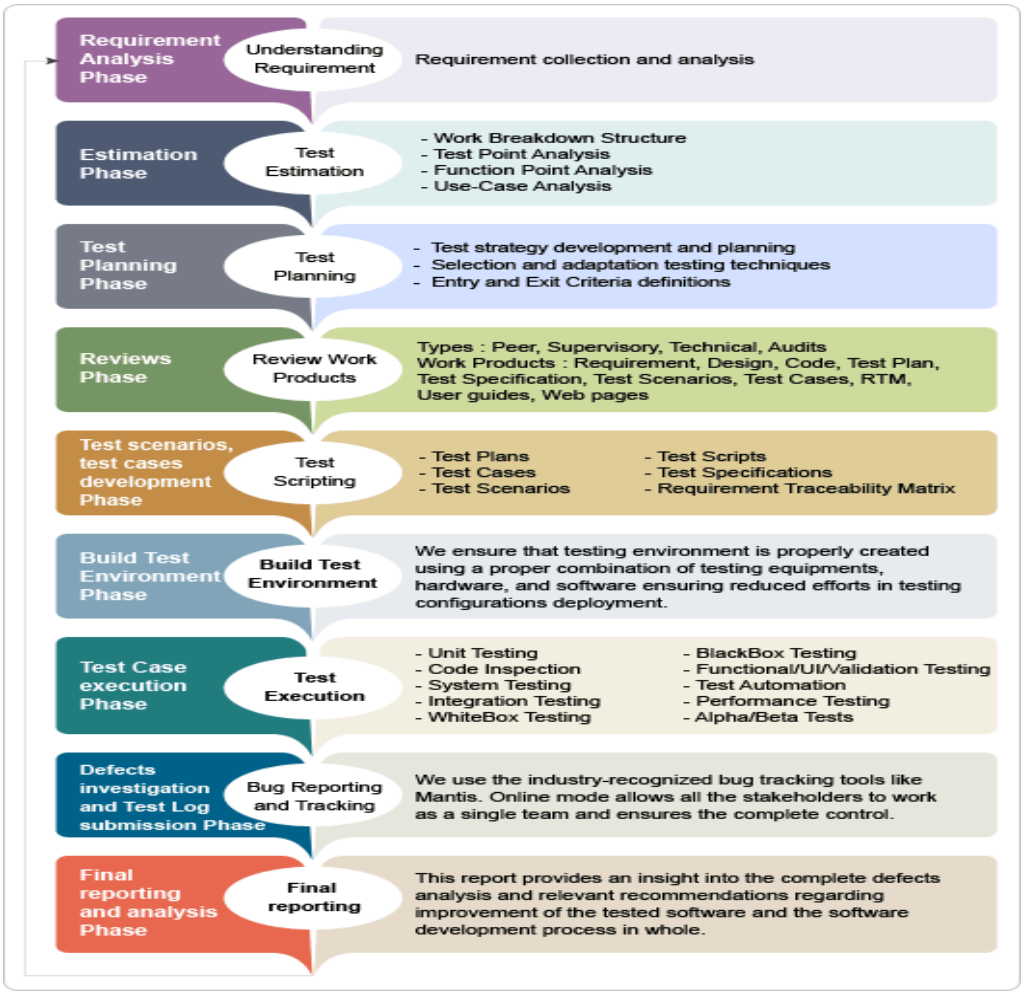
- It is important to **study** the **system** in depth in order to understand the system. It helps the debugger to construct different representations of systems that are to be debugged.
- **Backward analysis** of the problem traces the program backward from the location of failure message in order to identify the region of faulty code. You need to study the region of defect thoroughly to find the cause of defects.
- **Forward analysis** of the program involves tracking the program forward using breakpoints or print statements at different points in the program. It is important to focus on the region where the wrong outputs are obtained.
- You must use the **past experience** of the software to check for similar problems. The success of this approach depends on the expertise of the debugger.

3 a. Explain a typical testing life cycle with illustration.

Software testing is a process used to identify the correctness, completeness and quality of developed computer software.

The developer of the software can then check that the results produced by the software are in accord with his or her expectations.

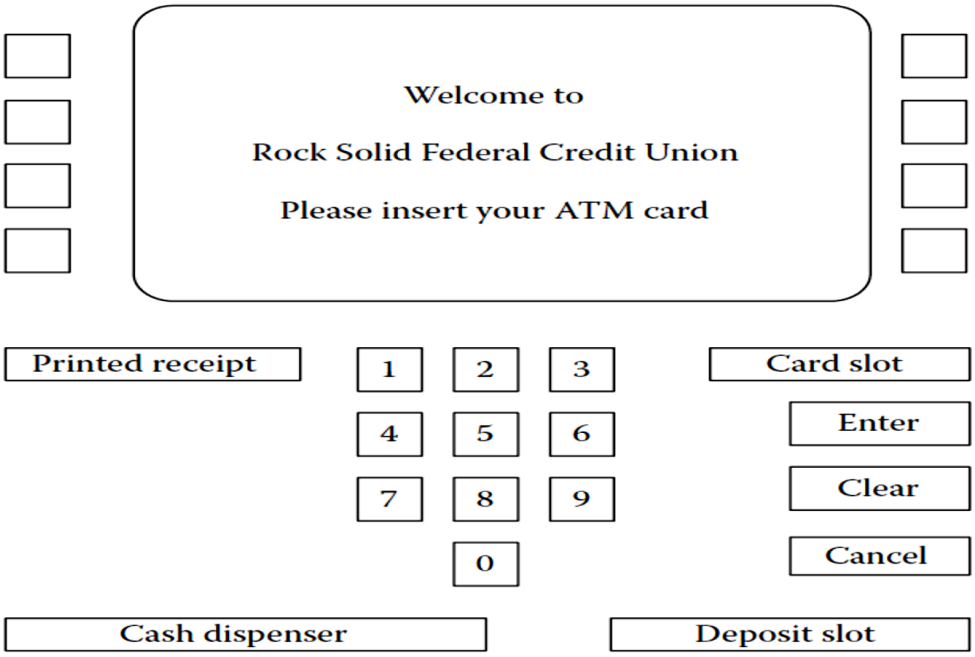

Error
Spec
Fault
Error
Design
Fault
Error
Coding
Fault
Testing
Incident
Classify
Isolate
Fix
Error
Fault
Putting Bugs IN
Development Phase
Finding Bugs
Testing Phase
Getting Bugs
OUT


Requirement Analysis Phase
Understanding Requirement
Requirement collection and analysis
Estimation Phase
Test Estimation
- Work Breakdown Structure
- Test Point Analysis
- Function Point Analysis
- Use-Case Analysis
Test Planning Phase
Test Planning
- Test strategy development and planning
- Selection and adaptation testing techniques
- Entry and Exit Criteria definitions
Reviews Phase
Review Work Products
Types : Peer, Supervisory, Technical, Audits
Work Products : Requirement, Design, Code, Test Plan, Test Specification, Test Scenarios, Test Cases, RTM, User guides, Web pages
Test scenarios, test cases development Phase
Test Scripting
- Test Plans
- Test Cases
- Test Scenarios
- Test Scripts
- Test Specifications
- Requirement Traceability Matrix
Build Test Environment Phase
Build Test Environment
We ensure that testing environment is properly created using a proper combination of testing equipments, hardware, and software ensuring reduced efforts in testing configurations deployment.
Test Case execution Phase
Test Execution
- Unit Testing
- Code Inspection
- System Testing
- Integration Testing
- WhiteBox Testing
- BlackBox Testing
- Functional/UI/Validation Testing
- Test Automation
- Performance Testing
- Alpha/Beta Tests
Defects investigation and Test Log submission Phase
Bug Reporting and Tracking
We use the industry-recognized bug tracking tools like Mantis. Online mode allows all the stakeholders to work as a single team and ensures the complete control.
Final reporting and analysis Phase
Final reporting
This report provides an insight into the complete defects analysis and relevant recommendations regarding improvement of the tested software and the software development process in whole.

3 b. Describe about SATM screen with problem statement.

Problem Statement

The SATM system communicates with bank customers via the 15 screens shown in Figure Using a terminal with features as shown in Figure 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.



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.

4.a State and Explain Data Flow Diagram for the triangle problem

Problem Statement

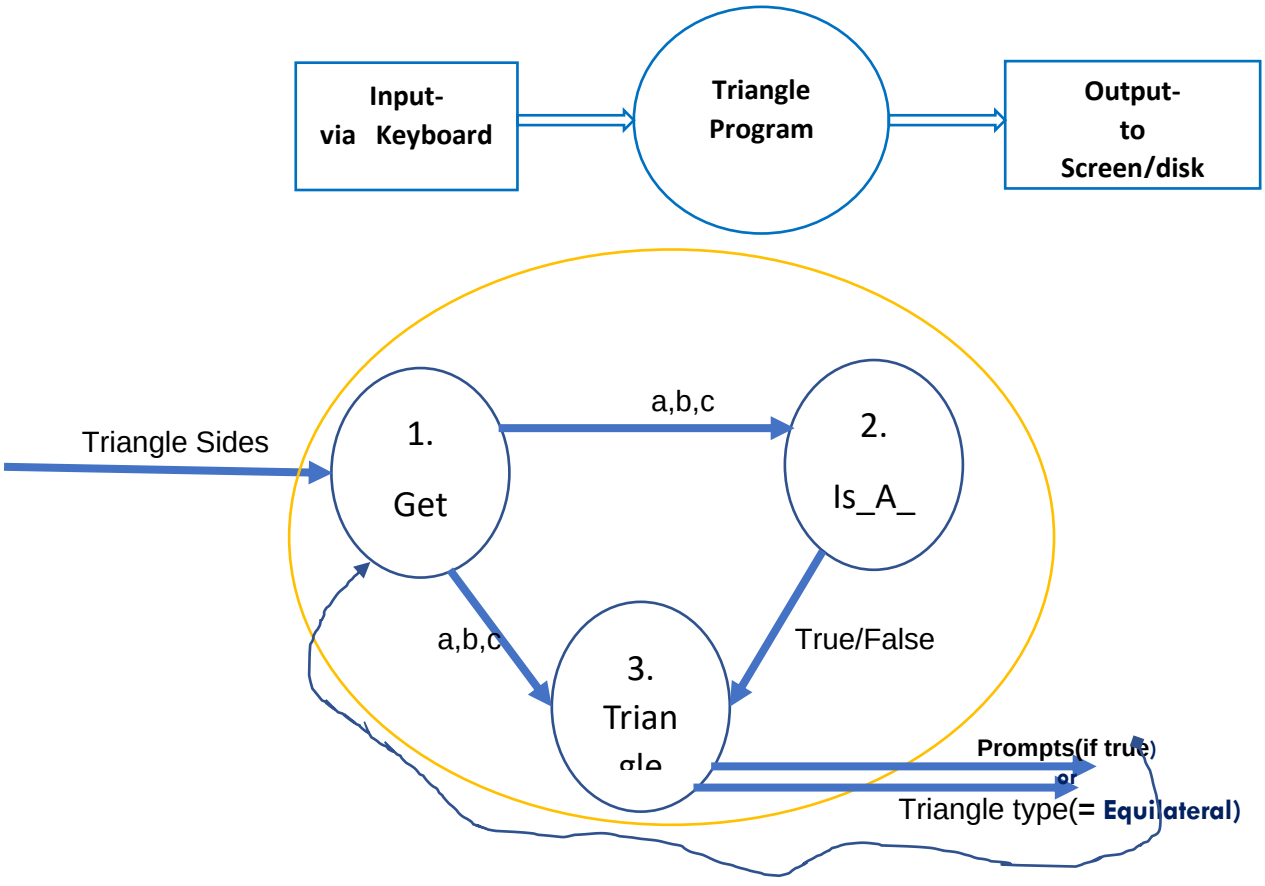
Simple version: The triangle program accepts three integers, a, b, and c, as input. These are taken to be sides of a triangle. The output of the program is the type of triangle determined by the three sides: *Equilateral, Isosceles, Scalene, or Not a Triangle.*

Improved version: The triangle program accepts three integers a, b and c must satisfy the following conditions:

c1. $1 \leq a \leq 200$ c4. $a < b + c$

c2. $1 \leq b \leq 200$ c5. $b < a + c$

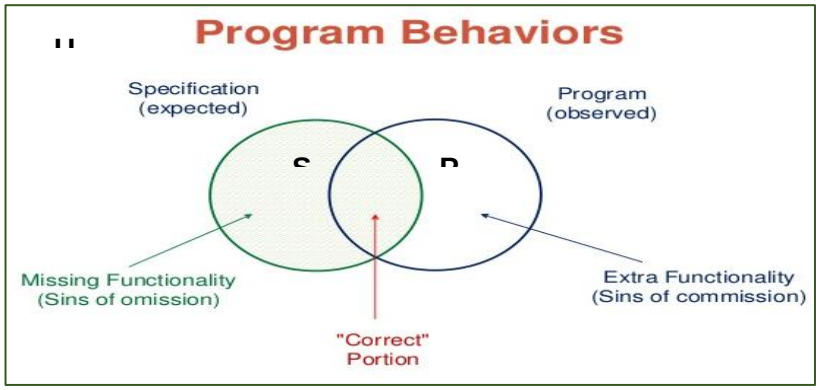
c3. $1 \leq c \leq 200$ c6. $c < a + b$

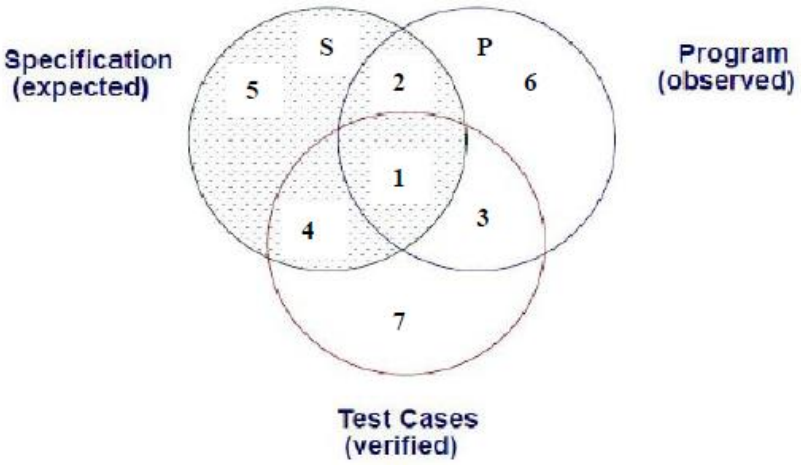


4.b. Explain Program behavior and tested behavior with Venn diagram

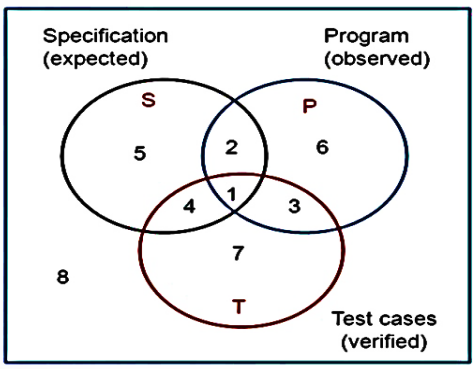
A Venn diagram is made up of two or more overlapping circles. It is often used in mathematics to show relationships between sets. In software testing, Venn Diagrams are used to represent *input/output relationships* for the system.

- Testing is basically concerned with behavior. Behavior is orthogonal to structural view. Structural view focuses on what it is and behavioral view considers what it does.
- One difficulty for testers is that documents are written for developers, so emphasis is on structural view instead of behavioral information. Simple Venn diagrams clarify several issues.





Structure and Functional View (Cont.)



- 2,5 Spec. but do not be tested
- 1,4 Spec. and Test
- 3,7 Test does not meet Spec.
- 2,6 Program is not tested
- 1,3 Program is under test
- 4,7 Test case do not have program.

4.c. Discuss Fault Taxonomy and give two example.

Input/Output Fault

Logic Fault

Computation Fault

Interface Fault

Data Fault

Type	*Instances*
Input	Correct input not accepted
	Incorrect input accepted
	Description wrong or missing
	Parameters wrong or missing
Output	Wrong format
	Wrong result
	Correct result at wrong time (too early, too late)
	Incomplete or missing result
	Spurious result
	Spelling/grammar
	Cosmetic

Missing case(s)
Duplicate case(s)
Extreme condition neglected
Misinterpretation
Missing condition
Extraneous condition(s)
Test of wrong variable
Incorrect loop iteration
Wrong operator (e.g., < instead of ≤)

5.a. Explain Boundary Value Analysis with example.

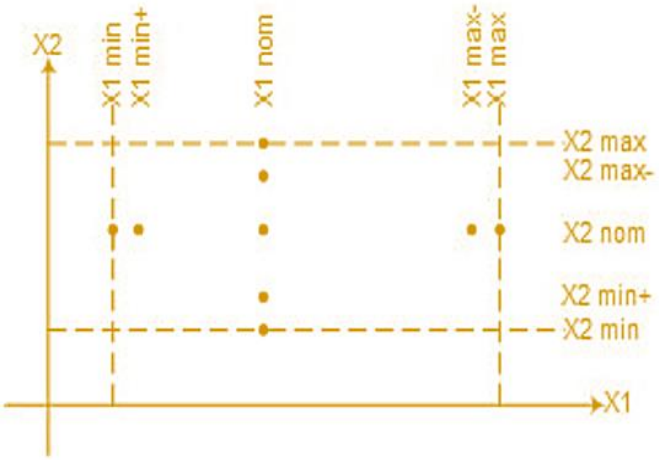
Boundary testing is the process of testing between extreme ends or boundaries between partitions of the input values. So these extreme ends like

- Start- End,
- Lower- Upper,
- Maximum-Minimum,
- Just Inside-Just Outside values

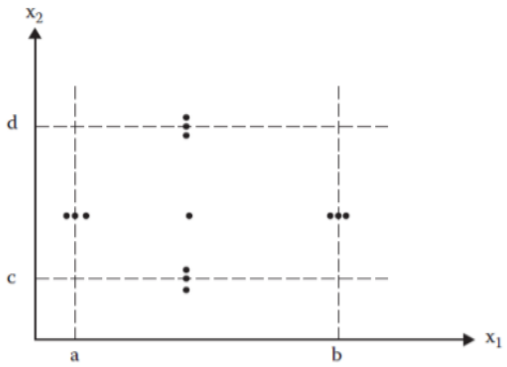
are called boundary values and the testing is called "boundary testing".

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



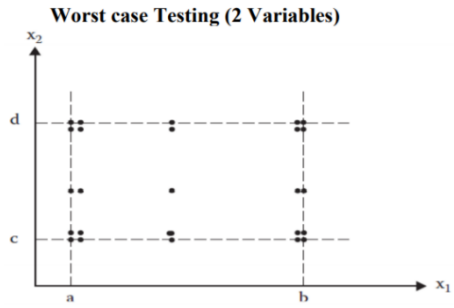
Robustness testing



. Robustness testing is a simple extension of boundary value Analysis. Here we add **min-** and **max+** values as additions.

• The boundary value analysis applies directly to robustness testing, especially the generalizations and limitations. The most interesting part of the robustness testing is not with the inputs, but with the **expected outputs**. • The main value of robustness testing is that it forces attention on exception handling.

Worst case testing



Boundary value analysis makes a single fault assumption of reliability theory.

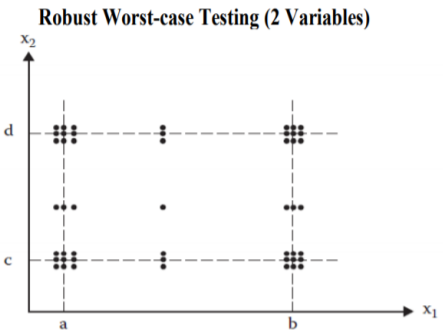
• Rejecting single fault assumption, we are interested in what happens when more than one variable has an extreme value. This is called worst case analysis.

• For each variable, we start with the **five elements set** that contain the ***min, min+, nom, max-, and max*** values.

• We then take the Cartesian product of the sets to generate test cases. Worst – case testing for a function of n variables generates 5n test cases.

• In this case $5^2 = 5*5 = 25$ test cases

Robust Worst-case Testing



Worst-case testing follows the generalization pattern of boundary value analysis.

• It also has the same limitations, particularly those related to independence.

• The best application for worst-case testing is where physical variables have numerous interactions and where failure of the function is extremely costly.

• For really paranoid testing, we do robust worst- case testing. This involves Cartesian product of the **seven - elements sets** we used in robustness testing resulting in 7n test cases. Min, min+, min-, max, max+, max-, nom

Here we have $7^2 = 7*7 = 49$ test cases

5.b. Equivalence Class Test Cases for the NextDate Function

The NextDate function illustrates very well the craft of choosing the underlying equivalence relation. NextDate is a function of three variables month, day, and year and these have intervals of valid classes defined as follows:

M1 = {month: 1≤ month ≤ 12}

D1 = {day: 1≤day≤31}

Y1 = {year: 1812≤year≤2012}

The invalid equivalence classes are:

M2 = {month : month < 1

M3 = {month : month > 12}

D2 = {day : day < 1}

D3 = {day : day > 31}
Y2 = {year : year < 1812}
Y3 = {year : year > 2012}
The number of valid classes equals the number of independent variables, only one weak normal equivalence class test case occurs, and it is identical to the strong normal equivalence class test case.

Weak /Strong normal equivalence

Case ID	*Month*	*Day*	*Year*	*Expected Output*
WN1, SN1	6	15	1912	6/16/1912

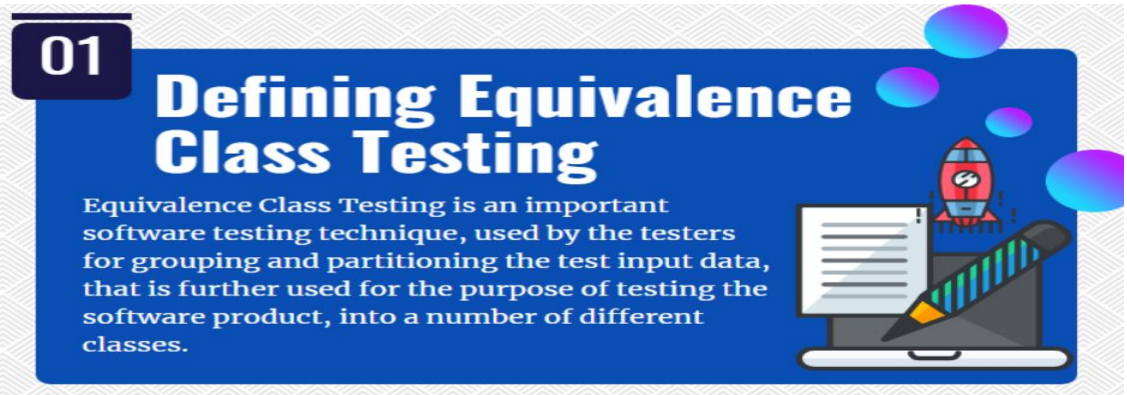
Weak Robust Test Cases

Case ID	*Month*	*Day*	*Year*	*Expected Output*
WR1	6	15	1912	6/16/1912
WR2	–1	15	1912	Value of month not in the range 1 ... 12
WR3	13	15	1912	Value of month not in the range 1 ... 12
WR4	6	–1	1912	Value of day not in the range 1 ... 31
WR5	6	32	1912	Value of day not in the range 1 ... 31
WR6	6	15	1811	Value of year not in the range 1812 ... 2012
WR7	6	15	2013	Value of year not in the range 1812 ... 2012

Strong robust equivalence

Case ID	*Month*	*Day*	*Year*	*Expected Output*
SR1	–1	15	1912	Value of month not in the range 1 ... 12
SR2	6	–1	1912	Value of day not in the range 1 ... 31
SR3	6	15	1811	Value of year not in the range 1812 ... 2012
SR4	–1	–1	1912	Value of month not in the range 1 ... 12 Value of day not in the range 1 ... 31
SR5	6	–1	1811	Value of day not in the range 1 ... 31 Value of year not in the range 1812 ... 2012
SR6	–1	15	1811	Value of month not in the range 1 ... 12 Value of year not in the range 1812 ... 2012
SR7	–1	–1	1811	Value of month not in the range 1 ... 12 Value of day not in the range 1 ... 31 Value of year not in the range 1812 ... 2012

6.a. What is the equivalence class test? What are different form equivalence class test?



Equivalence Partitions analysis

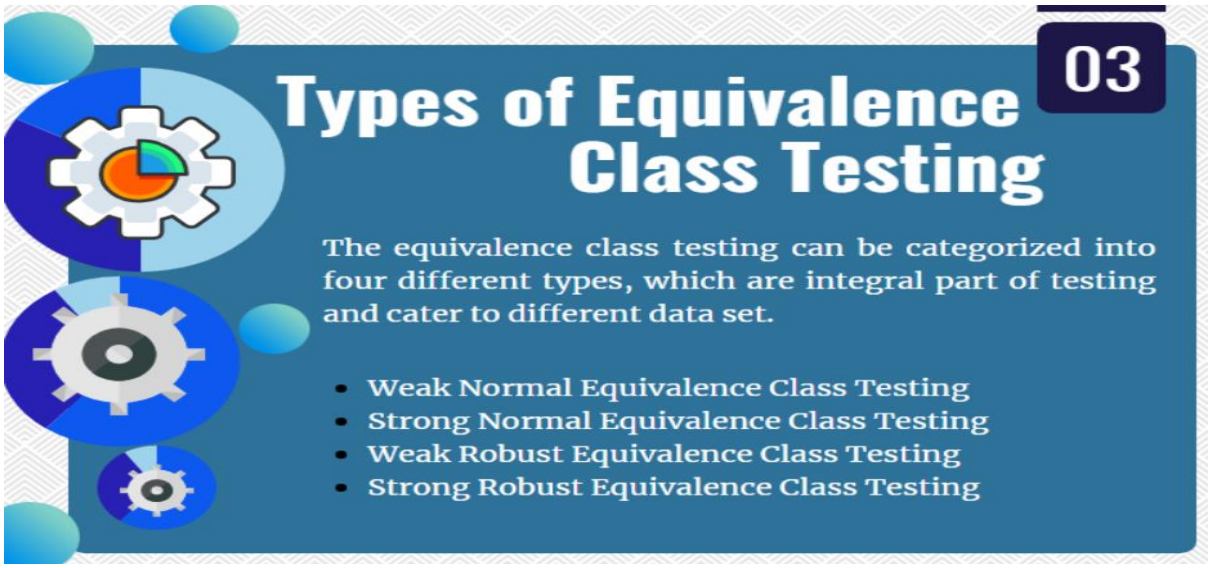
Example: Requirement for 'Password' field from "Add users modal window" of Admins actions menu: password field can not be shorter than 4 and longer than 28 (including) characters (numeric and alphabetic)

Equivalance classes

2 ↓	15 ↓	35 ↓
< 4	between 4 and 28	>28
invalid	valid	invalid

Define and execute the test cases:

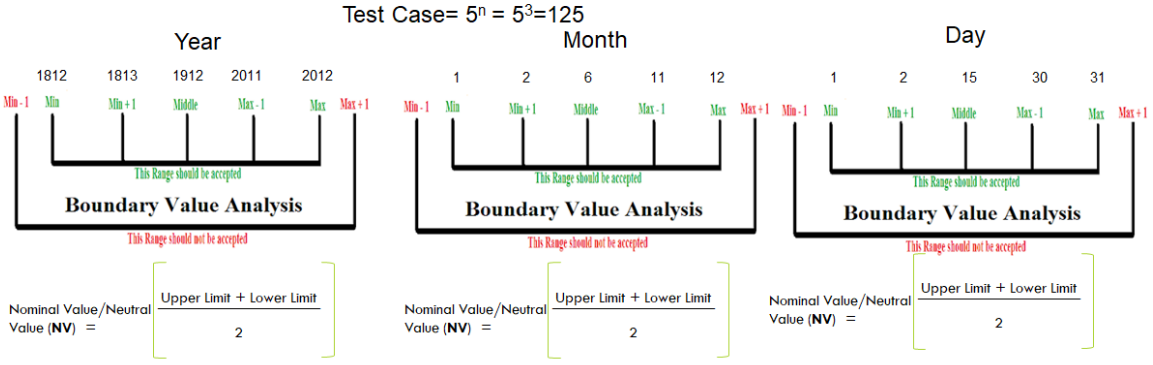
1. Password field contain 2 characters – Fail;
2. Password field contain 15 characters – Pass;
3. Password field contain 35 characters – Fail.



6.b. BVA for Next date function

All 125 worst-case test cases for NextDate are listed in Table. Take some time to examine it for gaps of untested functionality and for redundant testing. For example, would anyone actually want to test January 1 in five different years? Is the end of February tested sufficiently?

Test Case= $5^n = 5^3=125$



NV=2012+1812/2=3824/2=1912

NV=12+1/2=13/2=6

NV=30+1/2=31/2=15

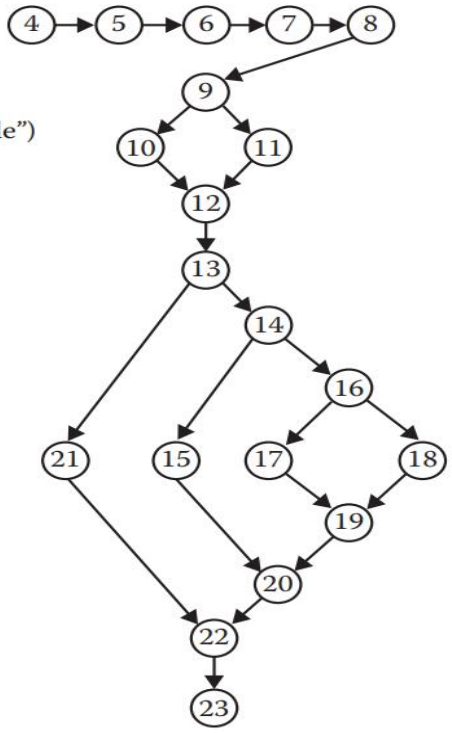
6.c. Decision table testing

Decision table testing is a software testing technique used to test system behavior for different input combinations. This is a systematic approach where the different input combinations and their corresponding system behavior (Output) are captured in a tabular form. That is why it is also called as a Cause-Effect table where Cause and effects are captured for better test coverage

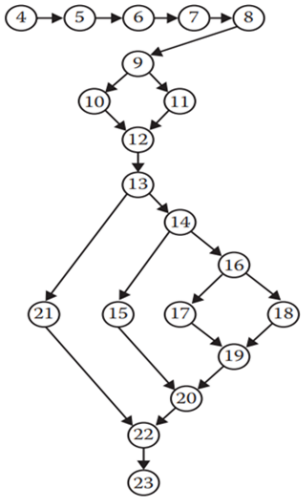
Case Id	Description	Input Data			Expected Output	Actual Output	Status
		a	b	c			
1	Enter the value of a, b and c such that a is less than sum of two sides	4	1	2	Message should be displayed can't form a triangle		
2	Enter the value of a, b and c such that b is less than sum of two sides and a is less than sum of other two sides	1	4	2	Message should be displayed can't form a triangle		
3	Enter the value of a, b and c such that c is less than sum of two sides and a and b is less than sum of other two sides	1	2	4	Message should be displayed can't form a triangle		
4	Enter the value a, b and c satisfying precondition and a=b, b=c and c=a	5	5	5	Should display the message Equilateral triangle		
5	Enter the value a ,b and c satisfying precondition and a=b and b ≠ c	2	2	3	Should display the message Isosceles triangle		
6	Enter the value a ,b and c satisfying precondition and b=c and c ≠ a	3	2	2	Should display the message Isosceles triangle		
7	Enter the value a ,b and c satisfying precondition and c=a and a ≠ b	2	3	2	Should display the message Isosceles triangle		
8	Enter the value a, b and c satisfying precondition and a ≠b , b ≠ c and c ≠ a	3	4	5	Should display the message Scalene triangle		

7.a. DD path for triangle problem.

```
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 Nodes	DD-Path Name	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



This is a complex definition, so we will apply it to the program graph.

– Node 4 is a **case 1** DD-Path; we will call it **first.**
– Similarly, node 23 is a **case 2** DD-Path; we will call it **last.**
– Nodes 5 through 8 are **case 5** DD-Paths. We know that node 8 is the last node in this DD-Path because it is the last node that preserves the 2-connectedness property of the chain.
– If we stop at node 7, we violate the "maximal" criterion.
– Nodes 10, 11, 15, 17, 18, and 21 are **case 4** DD-Paths.
– Nodes 9, 12, 13, 14, 16, 19, 20, and 22 are **case 3** DD-Paths.

7.b. Waterfall and 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

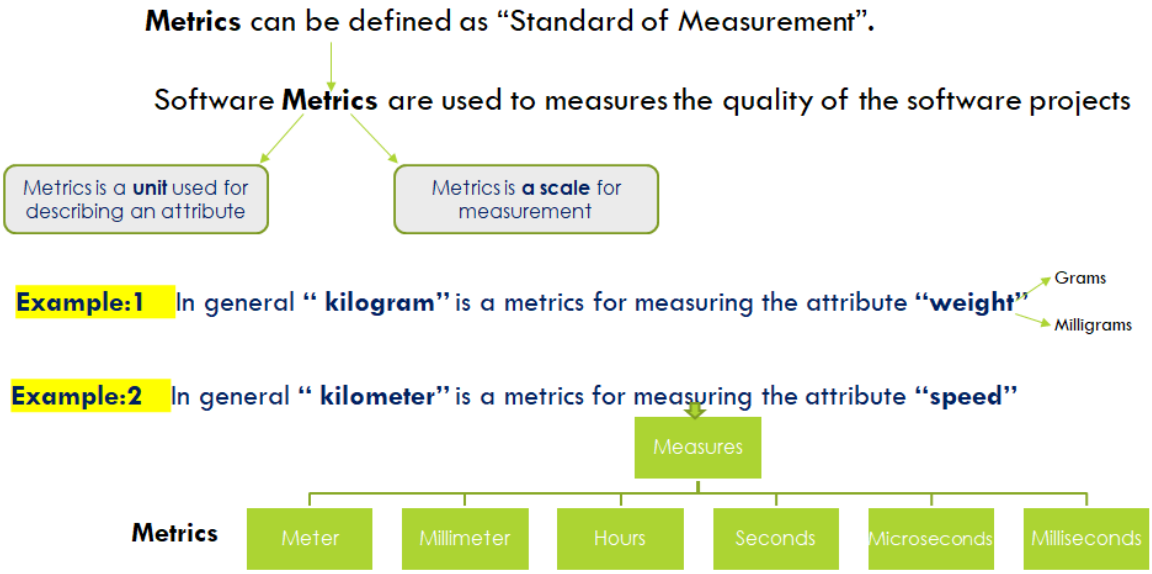
8a.Slice based testing and metric based testing

•**Program slice** is a set of program statements that contributes to, or affects the value of, a variable at some point in a program.
•We continue with the notation we used for define/use paths: a program P that has a program graph G(P) and a set of program variables V.

Definition:
•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 static, backward slice on the variable set V at statement fragment n, written S(V, n), is the set of node numbers of all statement fragments in P that contribute to the values of variables in V at statement fragment n

Program slices

1. **Backward slice** : Backward slices refer to statement fragments that contribute to the value of v at statement n.
2. **Forward slices:** refer to all the program statements that are affected by the value of v and statement n



8.b. Guideline for data flow testing

Data Flow Testing is a specific strategy of software testing that focuses on data variables and their values. It makes use of the control flow graph. When it comes to categorization Data flow testing will can be considered as a type of white box testing and structural types of testing. It keeps a check at the data receiving points by the variables and its usage points. It is done to cover the path testing and branch testing gap.
The process is conducted to detect the bugs because of the incorrect usage of data variables or data values. For e.g. Initialization of data variables in programming code, etc.

What is Data flow Testing?

- The programmer can perform numerous tests on data values and variables. This type of testing is referred to as data flow testing.
- It is performed at two abstract levels: static data flow testing and dynamic data flow testing.
- The static data flow testing process involves analyzing the source code without executing it.
- Static data flow testing exposes possible defects known as data flow anomaly.
- Dynamic data flow identifies program paths from source code.

Let us understand this with the help of an example.

1. read x;	
2. If(x>0)	(1, (2, t), x), (1, (2, f), x)
3. a= x+1	(1, 3, x)
4. if (x<=0) {	(1, (4, t), x), (1, (4, f), x)
5. if (x<1)	(1, (5, t), x), (1, (5, f), x)
6. x=x+1; (go to 5)	(1, 6, x)
else	
7. a=x+1	(1, 7, x)
8. print a;	(6,(5, f)x), (6,(5,t)x)
	(6, 6, x)
	(3, 8, a), (7, 8, a).

There are 8 statements in this code. In this code we cannot cover all 8 statements in a single path as if 2 is valid then **4, 5, 6, 7** are not traversed, and if 4 is valid then statement 2 and 3 will not be traversed.

Hence we will consider two paths so that we can cover all the statements.

x= 1
Path – 1, 2, 3, 8
Output = 2

If we consider **x = 1**, in step 1; x is assigned a value of 1 then we move to step 2 (since, x>0 we will move to statement **3 (a= x+1)** and at end, it will go to statement 8 and print x =2.

For the second path, we assign x as 1

Set x= -1
Path = 1, 2, 4, 5, 6, 5, 6, 5, 7, 8

Output = 2

x is set as 1 then it goes to step 1 to assign x as 1 and then moves to step 2 which is false as x is smaller than 0 (x>0 and here x=-1). It will then move to step 3 and then jump to step 4; as 4 is true (x<=0 and their x is less than 0) it will jump on 5 (x<1) which is true and it will move to step 6 **(x=x+1)** and here x is increased by 1.

So,

x=-1+1

x=0

x become 0 and it goes to step 5(x<1),as it is true it will jump to step

6 (x=x+1)

x=x+1

x= 0+1

x=1

x is now 1 and jump to step 5 (x<1) and now the condition is false and it will jump to step 7 **(a=x+1)** and set a=2 as x is 1. At the end the value of a is 2. And on step 8 we get the output as 2.

Steps of Data Flow Testing

- creation of a data flow graph.
- Selecting the testing criteria.
- Classifying paths that satisfy the selection criteria in the data flow graph.
- Develop path predicate expressions to derive test input.

The life cycle of data in programming code

- Definition: it includes defining, creation and initialization of data variables and the allocation of the memory to its data object.
- Usage: It refers to the user of the data variable in the code. Data can be used in two types as a predicate(P) or in the computational form(C).
- Deletion: Deletion of the Memory allocated to the variables.

Types of Data Flow Testing

- Static Data Flow Testing

No actual execution of the code is carried out in Static Data Flow testing. Generally, the definition, usage and kill pattern of the data variables is scrutinized through a control flow graph.

- Dynamic Data Flow Testing

The code is executed to observe the transitional results. Dynamic data flow testing includes:

- Identification of definition and usage of data variables.
- Identifying viable paths between definition and usage pairs of data variables.
- Designing & crafting test cases for these paths.

Advantages of Data Flow Testing

- Variables used but never defined,
- Variables defined but never used,
- Variables defined multiple times before actually used,
- DE allocating variables before using.

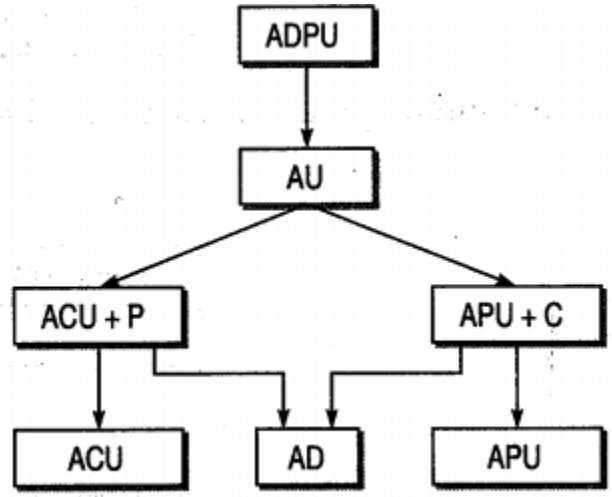
Data Flow Testing Limitations

- Testers require good knowledge of programming.
- Time-consuming
- Costly process.

Data Flow Testing Coverage

- All definition coverage: Covers “sub-paths” from each definition to some of their respective use.
- All definition-C use coverage: “sub-paths” from each definition to all their respective C use.
- All definition-P use coverage: “sub-paths” from each definition to all their respective P use.
- All use coverage: Coverage of “sub-paths” from each definition to every respective use irrespective of types.
- All definition use coverage: Coverage of “simple sub-paths” from each definition to every respective use.

Data Flow Testing Strategies



Following are the test selection criteria

1. All-defs: For every variable x and node i in a way that x has a global declaration in node I, pick a comprehensive path including the def-clear path from node i to
   - Edge (j,k) having a p-use of x or
   - Node j having a global c-use of x
2. All c-uses: For every variable x and node i in a way that x has a global declaration in node i, pick a comprehensive path including the def-clear path from node i to all nodes j having a global c-use of x in j.
3. All p-uses: For every variable x and node i in a way that x has a global declaration in node i, pick a comprehensive path including the def-clear path from node i to all edges (j,k) having p-use of x on edge (j,k).
4. All p-uses/Some c-uses: it is similar to all p-uses criterion except when variable x has no global p-use, it reduces to some c-uses criterion as given below
5. Some c-uses: For every variable x and node i in a way that x has a global declaration in node i, pick a comprehensive path including the def-clear path from node i to some nodes j having a global c-use of x in node j.
6. All c-uses/Some p-uses:it is similar to all c-uses criterion except when variable x has no global c-use, it reduces to some p-uses criterion as given below:
7. Some p-uses: For every variable x and node i in a way that x has a global declaration in node i, pick a comprehensive path including def-clear paths from node i to some edges (j,k) having a p-use of x on edge (j,k).
8. All uses:it is a combination of all p-uses criterion and all c-uses criterion.
9. All du-paths:For every variable x and node i in a way that x has a global declaration in node i, pick a comprehensive path including all du-paths from node i
   - To all nodes j having a global c-use of x in j and
   - To all edges (j,k) having a p-use of x on (j,k).

Data Flow Testing Applications

As per studies defects identified by executing **90%** "data coverage" is twice as compared to bugs detected by **90%** branch coverage.

The process flow testing is found effective, even when it is not supported by automation.

It requires extra record keeping; tracking the variables status. The computers help easy tracking of these variables and hence reducing the testing efforts considerably. Data flow testing tools can also be integrated into compilers.

9.a. Explain Mutation analysis and fault-based adequacy criteria.

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

Mutation Analysis: Terminology

Original program under test: The program or procedure (function) to be tested.

Mutant: A program that differs from the original program for one syntactic element (e.g., a statement, a condition, a variable, a label).

Distinguished mutant: A mutant that can be distinguished for the original program by executing at least one test case.

Equivalent mutant: A mutant that cannot be distinguished from the original program.

Mutation operator: A rule for producing a mutant program by syntactically modifying the original program.

Given a program and a test suite *T*, mutation analysis consists of the following steps:

Select mutation operators: If we are interested in specific classes of faults, we may select a set of mutation operators relevant to those faults.

Generate mutants: Mutants are generated mechanically by applying mutation operators to the original program.

Distinguish mutants: Execute the original program and each generated mutant with the test cases in T . A mutant is killed when it can be distinguished from the original program.

Figure 3 shows a sample of mutants for program Transduce, obtained by applying the mutant operators in **Figure 2.** Test suite TS

$$TS = \{1U, 1D, 2U, 2D, 2M, End, Long, Mixed\}$$

kills M_j, which can be distinguished from the original program by test cases $1D$, $2U$, $2D$, and $2M$. Mutants M_i, M_k, and M_l are not distinguished from the original program by any test in *TS*. We say that mutants not killed by a test suite are *live*.

9.b. Analysis and Test Plan

structure of an analysis and test plan is more standardized. A typical structure of a test and analysis plan includes information about items to be verified, features to be tested, the testing approach, pass and fail criteria, test deliverables, tasks, responsibilities and resources, and environment constraints. Basic elements are described in the sidebar on

A Standard Organization of an Analysis and Test Plan

Analysis and test items:
The items to be tested or analyzed. The description of each item indicates version and installation procedures that may be required.

Features to be tested:
The features considered in the plan.

Features not to be tested:
Features not considered in the current plan.

Approach:
The overall analysis and test approach, sufficiently detailed to permit identification of the major test and analysis tasks and estimation of time and resources.

Pass/Fail criteria:
Rules that determine the status of an artifact subjected to analysis and test.

Suspension and resumption criteria:
Conditions to trigger suspension of test and analysis activities (e.g., an excessive failure rate) and conditions for restarting or resuming an activity.

Risks and contingencies:
Risks foreseen when designing the plan and a contingency plan for each of the identified risks.

Deliverables:
A list all A&T artifacts and documents that must be produced.

Task and schedule:
A complete description of analysis and test tasks, relations among them, and relations between A&T and development tasks, with resource allocation and constraints. A task schedule usually includes GANTT and PERT diagrams.

Staff and responsibilities:
Staff required for performing analysis and test activities, the required skills, and the allocation of responsibilities among groups and individuals. Allocation of resources to tasks is described in the schedule.

Environmental needs:
Hardware and software required to perform analysis or testing activities.

10.a. **Scaffolding**

To do Unit tests, we have to provide replacements for parts of the program that we will omit from the test.

- *Scaffolding* is any code that we write, not as part of the application, but simply to support the process of Unit and Integration testing.
    - Scaffolding comes in two forms
        - Drivers
        - Stubs
- The purposes of scaffolding are to provide **controllability to execute test cases and observability to judge the outcome of test execution**. Sometimes scaffolding is

required to simply make a **module executable,** but even in incremental development with **immediate integration of each module**.

- **Generic versus Specific Scaffolding**
    - The simplest form of scaffolding is a driver program that runs a single, specific test case. If, for example, a test case specification calls for executing method calls in a particular sequence, this is easy to accomplish by writing the code to make the method calls in that sequence.
    - Fully generic scaffolding may be sufficient for small numbers of hand-written test cases. For larger test suites and particularly for those that are generated systematically, writing each test case by hand is impractical.

10.b. Test Oracles

Software that applies a pass/fail criterion to a program execution is called a test oracle.

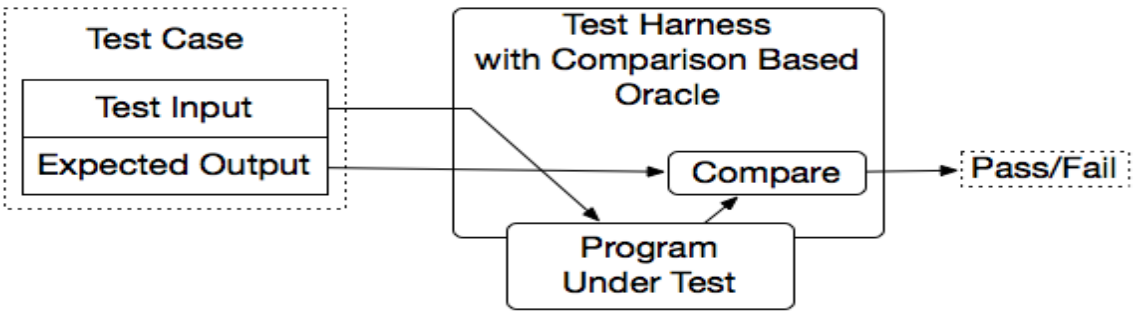
- A test oracle would classify every execution of a correct program as passing and would detect every program failure. In practice, the pass/fail criterion is usually imperfect.
- A test oracle may apply a pass/fail criterion that reflects only part of the actual program specification, and therefore passes some program executions, it has to fail.
- A test oracle may also give false alarms, failing an execution that it has to pass.
- False alarms in test execution are highly undesirable because of the direct expense of manually checking them and real failures will be overlooked.
- The best we can obtain is an oracle that detects deviations from expectation that may or may not be actual failures.

Support for **comparison-based test oracles** is often included in a test harness program or testing framework as in **figure**. A harness typically takes two inputs:

(1) **the input to the program under test and**

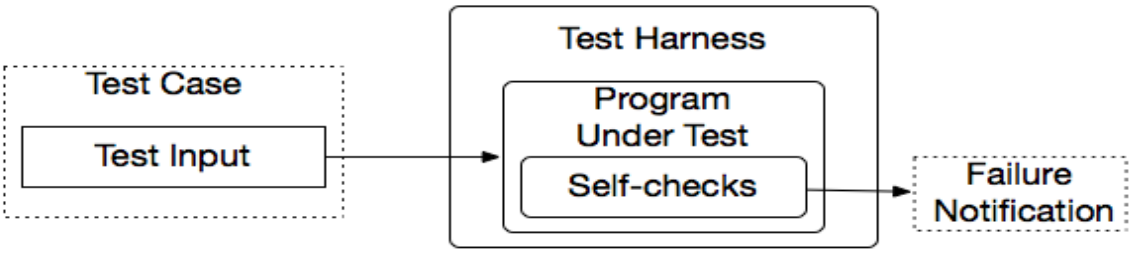
(2) **(2) the predicted output.** Frameworks for writing test cases as program code likewise provide support for comparison-based oracles.

A test harness with a comparison-based test oracle processes test cases consisting of (program input, predicted output) pairs.



A **program or module** specification describes all correct program behaviors, so an oracle based on a specification need not be paired with a particular test case. Instead, the oracle can be incorporated into the program under test. In general these self-check are in the form of **assertions**, similar to assertions used in symbolic execution and program verification but designed to be checked during execution.

When self-checks are embedded in the program, test cases need not include predicted outputs.



-