CMR INSTITUTE OF TECHNOLOGY

USN | | | | | | | | | |

Internal Assesment Test - III

Sub: Software Testing Code: 18MCA351

Date: 16.1.2020 Duration: 90 mins Max Marks: 50 Sem: III Branch: MCA

Answer Any One FULL Question from each part.

		Marks	OBE CO	OBE RBT
	Part - I			
1 (a)	Explain the basic principles of Analysis and testing and describe adequacy criteria and comparison criteria.	[10]	CO1	L1
	OR			
2 (a)	Discuss Waterfall Spin-off Specification based life cycle model.	[10]	CO1	L1
	Part – II			
3 (a)	List out quality attributes of software and explain each of them	[5]	CO1	L1
(b)	Differentiate between verification and validation	[5]	CO1	L3
	OR			
4 (a)	How is hardware testing different from software testing?	[5]	CO1	L2
(b)	Discuss the defect life-cycle and draw an appropriate diagram	[5]	CO1	L3
	ART - III			
5 (a)	Discuss different types of test metrics.	[6]	CO1	L2
(b)	Discuss the assumptions in fault based testing.	[4]	CO6	L2
	OR			
6 (a)	Illustrate mutation analysis with its variants and assumptions.	[10]	CO5	L2
	Part – IV			
7 (a)	Differentiate between unit testing, integration testing and system testing.	[10]	CO1	L2
	OR			
8 (a)	Explain Scaffolding. Compare between generic versus specific scaffolding	[10]	CO5	L2
	Part – V			
9 (a)	Explain: Self-checks as test oracle Capture & Replay	[10]	CO5	L2
	OR			
10(a)	Write short notes on Risk Planning Basis Path Testing	[10]	CO5	L2

1 a) Explain the basic principles of Analysis and testing and describe adequacy criteria and comparison criteria.

Ans:

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

A software test adequacy criterion is a predicate that defines what properties of a program must be exercised to constitute a thorough test. If the system passes an adequate suite of test cases, then it must be correct (or dependable). But determining an adequate suite of test case is hypothetical.

Use of adequacy criteria:

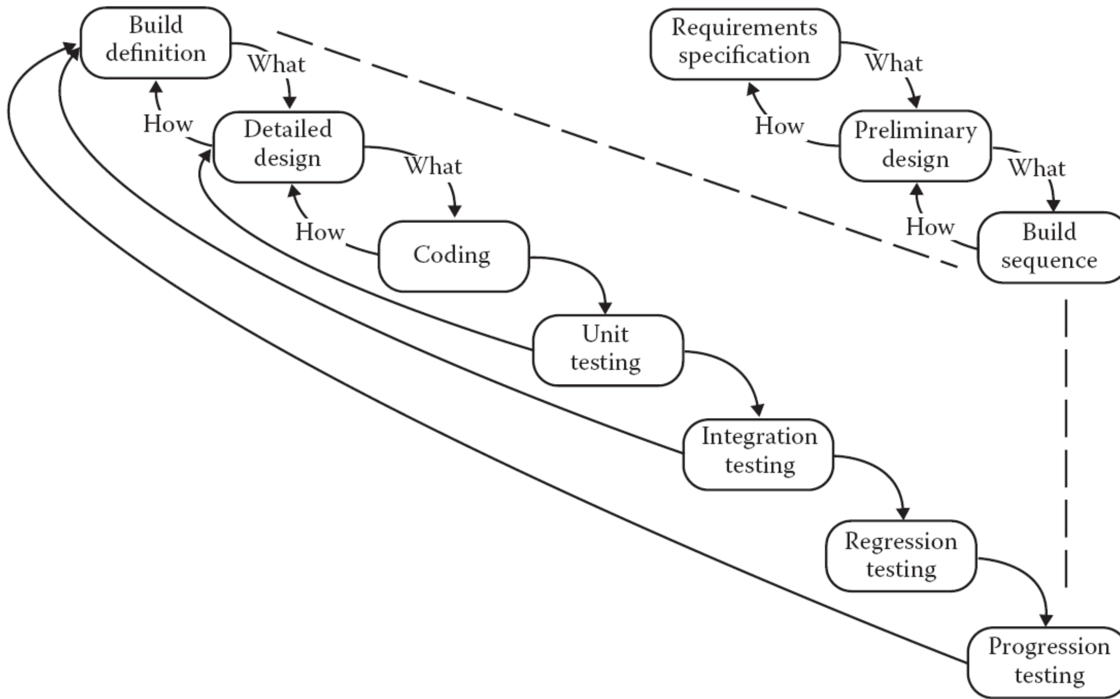
- Specify a software testing requirement

  -Determine test cases to satisfy requirement
- Determine observations that should be made during testing
- Control the cost of testing

  -Avoid redundant and unnecessary tests
- Help assess software dependability

Build confidence in the integrity estimate

2 a) Discuss Waterfall Spin-off Specification based life cycle model.
Ans: ***Waterfall Spin-Offs***
There are three mainline derivatives of the waterfall model: incremental development, evolutionary development, and the spiral model (Boehm, 1988). Each of these involves a series of increments or builds .It is important to keep preliminary design as an integral
phase rather than to try to amortize such high-level design across a series of builds.
This single design step cannot be done in the evolutionary and spiral
models. This is also a major limitation of the bottom–up agile methods.
Within a build, the normal waterfall phases from detailed design through testing occur with
one important difference: system testing is split into two steps—regression and progression testing.
The main impact of the series of builds is that regression testing becomes necessary. The goal
of regression testing is to ensure that things that worked correctly in the previous build still work
with the newly added code. Regression testing can either precede or follow integration testing, or
possibly occur in both places. Progression testing assumes that regression testing was successful
and that the new functionality can be tested. (We like to think that the addition of new code represents
progress, not a regression.) Regression testing is an absolute necessity in a series of builds

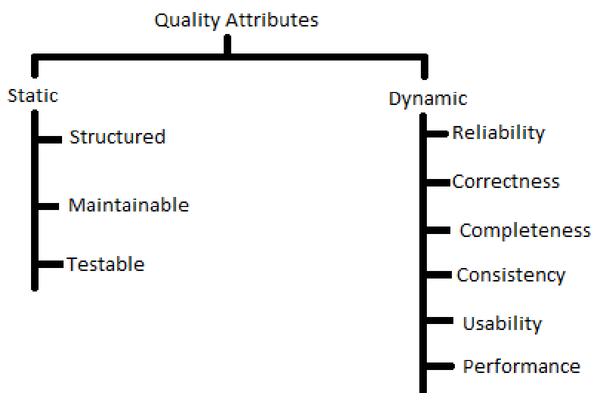


Evolutionary development is best summarized as client-based iteration. In this spin-off, a small initial version of a product is given to users who then suggest additional features. This is particularly helpful in applications for which time-to-market is a priority. The initial version might capture a segment of the target market, and then that segment is "locked in" to future evolutionary versions. When these customers have a sense that they are "being heard," they tend to be more invested in the evolving product.

Barry Boehm's spiral model has some of the flavor of the evolutionary model. The biggest difference is that the increments are determined more on the basis of risk rather than on client

3 a) List out quality attributes of software and explain each of them.

Ans:



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."

3 b) Differentiate between verification and validation.

Ans:

Verification	**Validation**
• Verifying process includes checking documents, design, code and program	• It is a dynamic mechanism of testing and validating the actual product
• It does ***not*** involve executing the code	• It always involves executing the code
• Verification uses methods like reviews, walkthroughs, inspections and desk- checking etc.	• It uses methods like black box testing ,white box testing and non-functional testing
• Whether the software conforms to specification is checked	• It checks whether software meets the requirements and expectations of customer
• It finds bugs early in the development cycle	• It can find bugs that the verification process can not catch
• Target is application and software architecture, specification, complete design, high level and data base design etc.	• Target is actual product
• QA team does verification and make sure that the software is as per the requirement in the SRS document.	• With the involvement of testing team validation is executed on software code.
• It comes before validation	• It comes after verification

4 a) How is hardware testing different from software testing?

Ans:

Software Product	**Hardware Product**
Does not degrade over time	Degrades over time
Fault present in application will remain and no new fault will creep in unless application is changed.	VLST chip might fail over time due to a fault that did not exist at the time chip was manufactured and tested.

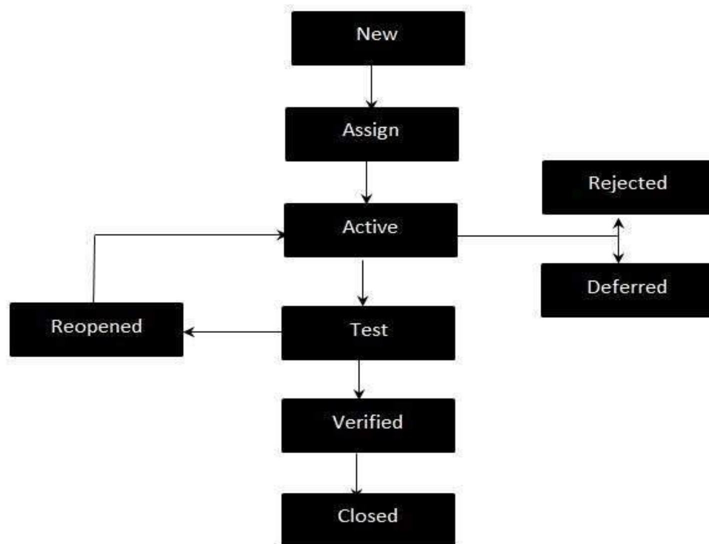
Built-in self test meant for hardware product, rarely can be applied to software design and code.	BIST intend to actually test for the correct functioning of a circuit
It only detects faults that were present when the last change was made	Hardware testers generate test based on fault models e.g Stuck-at fault model – one can use a set of input test patterns to test whether a logic gate is functioning as expected

4 b) Discuss the defect life-cycle and draw an appropriate diagram.
Ans:
Defect life cycle, also known as **Bug Life cycle** is the journey of a **defect cycle**, which a **defect** goes through during its lifetime. It varies from organization to organization and also from project to project as it is governed by the software testing process and also depends upon the tools used.

Defect Life Cycle States:

- **New -** Potential defect that is raised and yet to be validated.
- **Assigned -** Assigned against a development team to address it but not yet resolved.
- **Active -** The Defect is being addressed by the developer and investigation is under progress. At this stage there are two possible outcomes; viz - Deferred or Rejected.
- **Test -** The Defect is fixed and ready for testing.
- **Verified -** The Defect that is retested and the test has been verified by QA.
- **Closed -** The final state of the defect that can be closed after the QA retesting or can be closed if the defect is duplicate or considered as NOT a defect.
- **Reopened -** When the defect is NOT fixed, QA reopens/reactivates the defect.
- **Deferred -** When a defect cannot be addressed in that particular cycle it is deferred to future release.

Rejected - A defect can be rejected for any of the 3 reasons; viz - duplicate defect, NOT a Defect, Non Reproducible.



5 a) Discuss different types of test metrics.
Ans:

Why do Test Metrics?

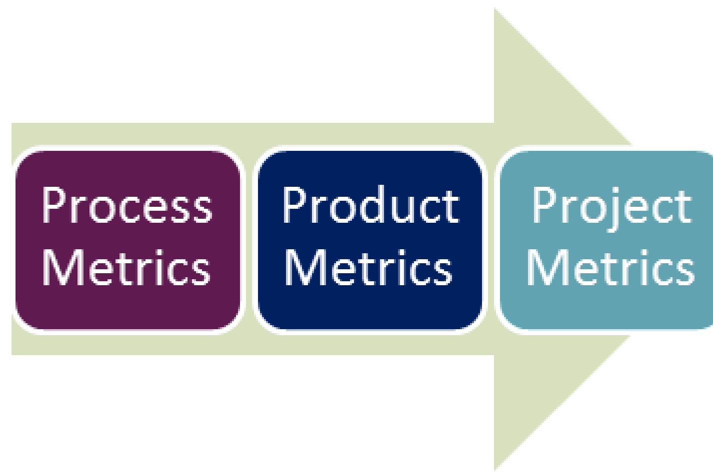
"We cannot improve what we cannot measure" and Test Metrics helps us to do exactly the same.

- Take decision for next phase of activities
- Evidence of the claim or prediction

- Understand the type of improvement required
- Take decision or process or technology change

Read more about its Importance of Test Metrics

Types of Metrics



- **Process Metrics:** It can be used to improve the process efficiency of the SDLC (Software Development Life Cycle)
- **Product Metrics:** It deals with the quality of the software product
- **Project Metrics:** It can be used to measure the efficiency of a project team or any testing tools being used by the team members

Identification of correct testing metrics is very important. Few things need to be considered before identifying the test metrics

- Fix the target audience for the metric preparation
- Define the goal for metrics
- Introduce all the relevant metrics based on project needs
- Analyze the cost benefits aspect of each metrics and the project lifestyle phase in which it results into the maximum output

5 b) Discuss the assumptions in fault based testing.
Ans:
The effectiveness of fault-based testing depends on the quality of the fault model, and on some basic assumptions about the relation of the seeded faults to faults that might actually be present. In practice the seeded faults are small syntactic changes, like replacing one variable reference by another in an expression, or changing a comparison from < to <=. We may hypothesize that these are representative of faults actually present in the program. Put another way, if the program under test has an actual fault, we may hypothesize that it differs from another, corrected program by only a small textual change. If so, then we need merely distinguish the program from all such small variants (by se- Δ competent lecting test cases for which either the original or the variant program fails) to ensure programmer hypothesis detection of all such faults. This is known as the competent programmer hypothesis, an assumption that the program under test is “close to” (in the sense of textual difference) a correct program.

6 a) Illustrate mutation analysis with its variants and assumptions.
Ans: **Mutation testing** (or **mutation analysis** or program **mutation**) is used to design new software tests and evaluate the quality of existing software tests. **Mutation testing** involves modifying a program in small ways.
Mutation analysis is the most common form of software fault-based testing. A fault model is used to produce hypothetical faulty programs by creating variants of the program under test. Variants are created by “seeding” faults, that is, by making a small change to the program under test following a pattern in the fault model. The patterns Δ mutation operator for changing program text are called mutation operators, and each variant program is Δ mutant called a mutant.

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

7 a) Differentiate between unit testing, integration testing and system testing.
Ans:
Unit Testing is a software testing technique by means of which individual units of software i.e. group of computer program modules, usage procedures and operating procedures are tested to determine whether they are suitable for use or not. It is a testing method using which every independent modules are tested to determine if there are any issue by the developer himself. It is correlated with functional correctness of the independent modules.
Unit Testing is defined as a type of software testing where individual components of a software are tested.
Unit Testing of software product is carried out during the development of an application. An individual component may be either an individual function or a procedure. Unit Testing is typically performed by the developer.

System Testing	Integration Testing
1. Testing the completed product to check if it meets the specification requirements.	1. Testing the collection and interface modules to check whether they give the expected result
2. Both functional and non-functional testing are covered like sanity, usability, performance, stress an load .	2.Only Functional testing is performed to check whether the two modules when combined give correct outcome.
3. It is a high level testing performed after integration testing	3. It is a low level testing performed after unit testing
4. It is a black box testing technique so no knowledge of internal structure or code is required	4. It is both black box and white box testing approach so it requires the knowledge of the two modules and the interface
5. It is performed by test engineers only	5. Integration testing is performed by developers as well test engineers
6. Here the testing is performed on the system as a whole including all the external interfaces, so any defect found in it is regarded as defect of whole system	6. Here the testing is performed on interface between individual module thus any defect found is only for individual modules and not the entire system
7. In System Testing the test cases are developed to simulate real life scenarios	7. Here the test cases are developed to simulate the interaction between the two module
8. The System testing covers many different testing types like sanity, usability, maintenance, regression, retesting and performance	8. Integration testing techniques includes big bang approach, top bottom , bottom to top and sandwich approach.

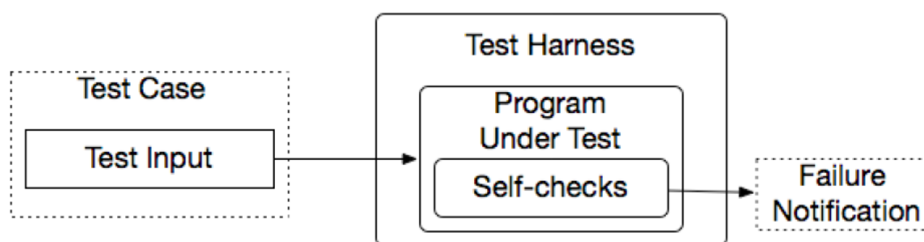
8 a) Explain Scaffolding. Compare between generic versus specific scaffolding.
Ans:
How general should scaffolding be? To answer
 We could build a driver and stubs for each test case or at least factor out some common code of the driver and test management (e.g., JUnit)

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

9 a)
Explain:
Ans: **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.



CAPTURE AND REPLAY
 Sometimes it is difficult to either devise a precise description of expected behaviour or adequately characterize correct behaviour for effective self checks.
Example: even if we separate testing program functionally from GUI, some testing of the GUI is required.
 If one cannot completely avoid human involvement test case execution, one can at least avoid unnecessary repetition of this cost and opportunity for error.
 The principle is simple:

The first time such a test case is executed, the oracle function is carried out by a human, and the interaction sequence is captured. Provided the execution was judged (by human tester) to be correct, the captured log now forms an (input, predicted output) pair for subsequent automated testing.
 The savings from automated retesting with a captured log depends on how many build-and-test cycles we can continue to use it, before it is invalidated by some change to the program.
 Mapping from concrete state to an abstract model of interacting sequences is some time possible but is generally quite limited.

10 a) Write short notes on
Risk Planning
Basis Path Testing
Ans:
Risk Planning:
Planning risks and contingencies

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

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

Late delivery of the software, hardware or tools.

Delays in training on the application and/or tools.

Changes to the original requirements or designs.

Complexities involved in testing the applications

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

The number of tests performed will be reduced.

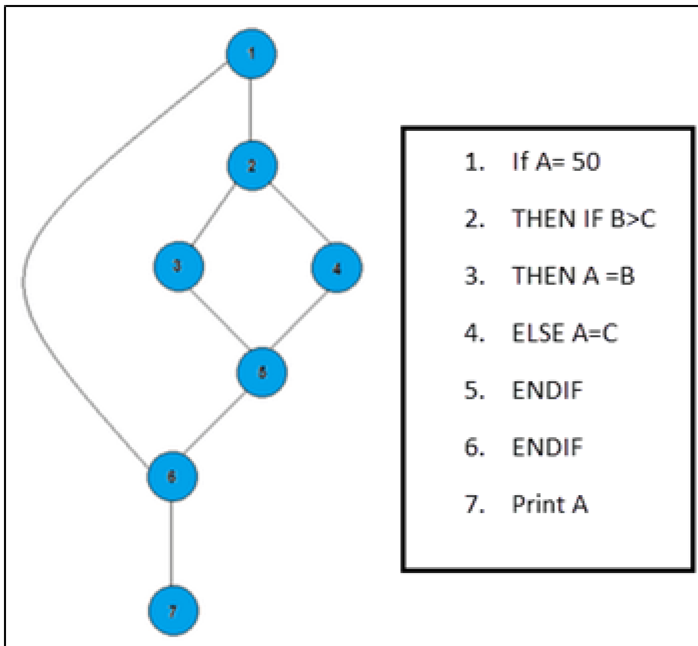
The number of acceptable defects will be increased.

Resources will be added to the test team.

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

The scope of the plan may be changed.

There may be some optimization of resources. This should be avoided, if possible, for obvious reasons.
Basis Path Testing:
The basis path testing is same, but it is based on a White Box Testing method, that defines test cases based on the flows or logical path that can be taken through the program. In software engineering, Basis path testing involves execution of all possible blocks in a program and achieves maximum path coverage with the least number of test cases. It is a hybrid of branch testing and path testing methods.
The objective behind basis path in software testing is that it defines the number of independent paths, thus the number of test cases needed can be defined explicitly (maximizes the coverage of each test case).
Here we will take a simple example, to get a better idea what is basis path testing include



In the above example, we can see there are few conditional statements that is executed depending on what condition it suffice. Here there are 3 paths or condition that need to be tested to get the output,

Path 1: 1,2,3,5,6, 7

Path 2: 1,2,4,5,6, 7

Path 3: 1, 6, 7

Steps for Basis Path testing

The basic steps involved in basis path testing include

Draw a control graph (to determine different program paths)

Calculate Cyclomatic complexity (metrics to determine the number of independent paths)

Find a basis set of paths

Generate test cases to exercise each path

Advantages of Basic Path Testing

It helps to reduce the redundant tests

It focuses attention on program logic

It helps facilitates analytical versus arbitrary case design

Test cases which exercise basis set will execute every statement in a program at least once

Conclusion:

Basis path testing helps to determine all faults lying within a piece of code.