STANNS COLLEGE OF ENGINEERING AND TECHNOLOGY
SOFTWARE TESTING METHODOLOGIES

Unit-4

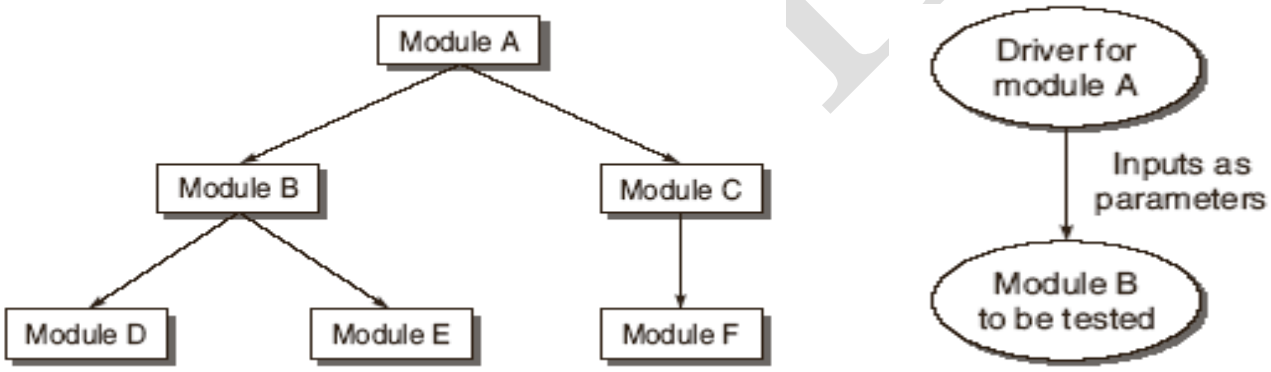
Validation Activities: Unit testing, Integration Testing, Function testing, System testing, Acceptance testing
Regression testing: Progressives Vs regressive testing, Regression testability, Objectives of regression testing, When regression testing done?, Regression testing types, Regression testing techniques

Validation Activities

Unit Testing:

A unit is the smallest testable part of an application like functions, classes, procedures, interfaces. Unit testing is a method by which individual units of source code are tested to determine if they are fit for use. Unit tests are basically written and executed by software developers to make sure that code meets its design and requirements and behaves as expected. The goal of unit testing is to segregate each part of the program and test that the individual parts are working correctly. This means that for any function or procedure when a set of inputs are given then it should return the proper values.

Drivers:



Drivers are also kind of dummy modules which are known as "calling programs", which is used when main programs are under construction. Suppose a module is to be tested, where in some inputs are to be received from another module. However this module which passes inputs to the module to be tested is not ready and under development. In such a situation, we need to simulate the inputs required in the module to be tested. This module where the required inputs for the module under test are simulated for the purpose of module or unit testing is known as **driver module**.

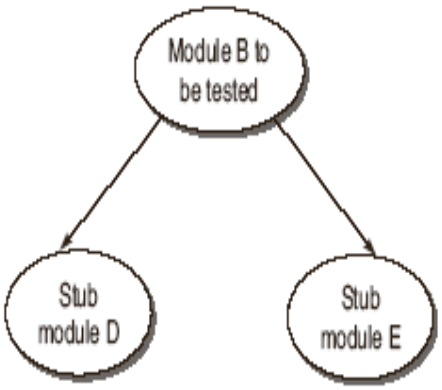
For example: When we have modules B and C ready but module A which calls functions from module B and C is not ready so developer will write a dummy piece of code for module A which will return values to module B and C. This dummy piece of code is known as driver.

Stubs:

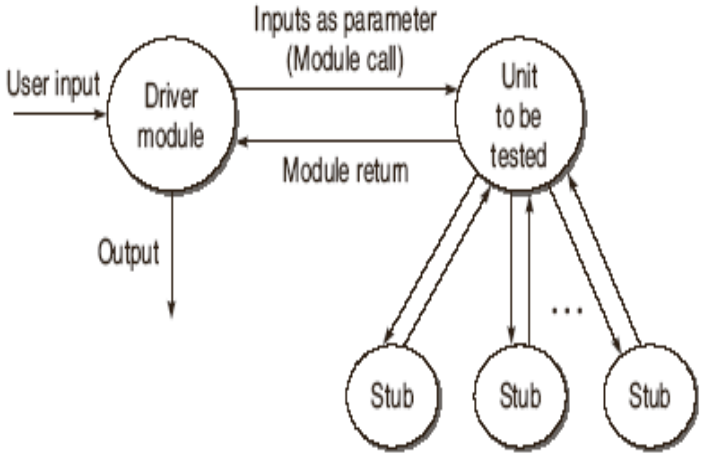
Stubs are dummy modules which are known as "called programs" which is used when sub programs are under construction. The module under testing may also call some other module which is not ready at the time of testing. Therefore, these modules need to be simulated for testing. In most cases, dummy modules instead of actual modules, which are not ready, are prepared for these

subordinate modules. These dummy modules are called **Stubs.**

Assume you have 3 modules, Module A, Module B and module C. Module A is ready and we need to test it, but module A calls functions from Module B and C which are not ready, so developer will write a dummy module which simulates B and C and returns values to module A. This dummy module code is known as stub.



Benifits of using Stubs and Drivers:



--Stubs allow the programmer to call a method in the code being developed, even if the method does not have the desired behaviour yet.
--By using stubs and drivers effectively, we can cut down our total debugging and testing small parts of a program individually, helping us to narrow down problems before they expand.
--Stubs and Drivers can also be an effective tool for demonstrating progress in a business environment.

Example:

```
main()
{
        int a,b,c,sum;
        scanf("%d%d",&a,&b);
        sum=calsum(a,b);
        diff=caldiff(a,b);
        mul=calmul(a,b);
        printf("The sum is %d:",sum);
}
calsum(int x, int y)
{
```

```
        int d;
        d=x+y;
        return d;
}
```

*Suppose if the main() module ot caldiff() & calmul() moduel is not ready, then the driver and stub module can be designed as:

Solution:

-->Driver for main() Module:

```
driver_main()
{
        int a,b,c,sum;
        scanf("%d%d",&a,&b);
        sum=calsum(a,b);
        printf("The sum is %d:",sum);
}
```

-->Stub for call_sum() module:

```
call_sum(int x, int y)
{
        printf("Difference calculating module")
        return 0;
}
```

Integration Testing:

Once all the individual units are created and tested, we start combining those "Unit Tested" modules and start doing the integrated testing. So the meaning of Integration testing is quite straight forward-Integrate/combine the unit tested module one by one and test the behaviour as a combined unit.

The main function or goal of Integration testing is to test the interfaces between the units/modules. The individual modules are first tested in isolation. Once the modules are unit tested, they are integrated one by one, till all the modules are integrated, to check the combinational behaviour, and validate whether the requirements are implemented correctly or not.

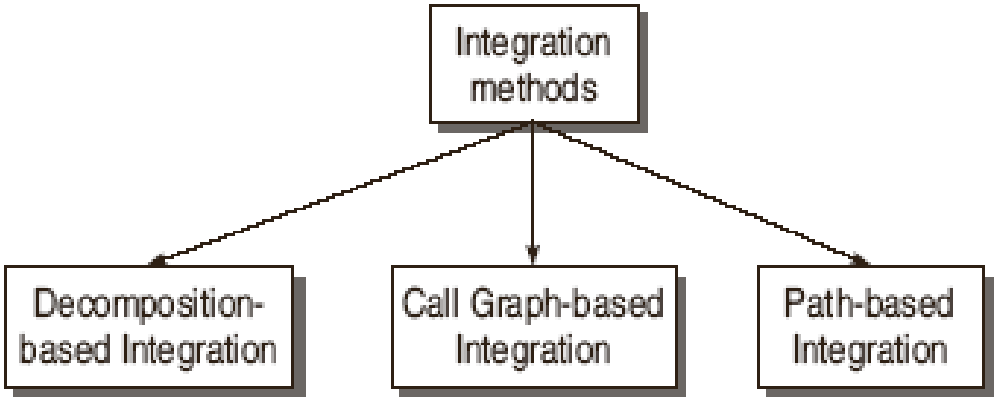
Integration Testing is necessary for the following reasons:
--It exposes inconsistency between the modules such as improper call or return sequences.

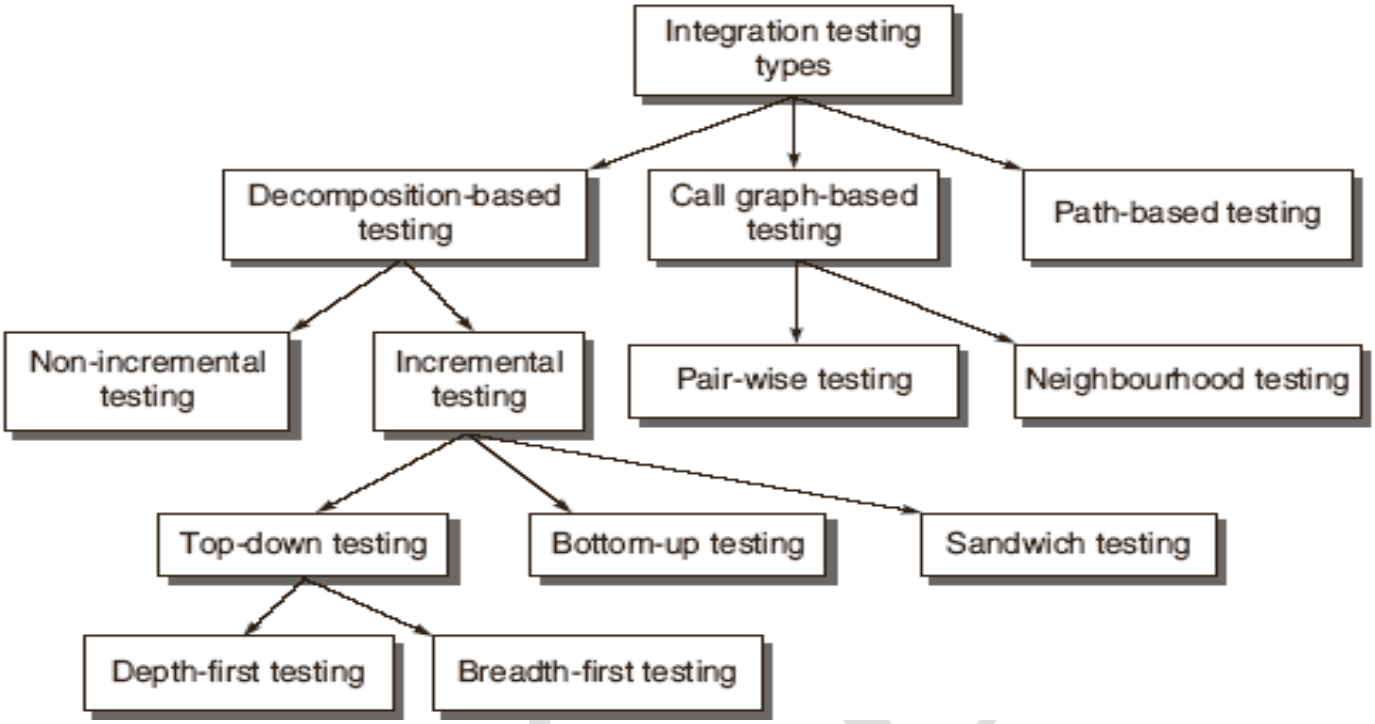
--Data can be lost across an interface.

--One module when combined with another module may not give the desired result.

--Data types and their valid ranges may mismatch between the modules.

There are three approaches for integration testing:



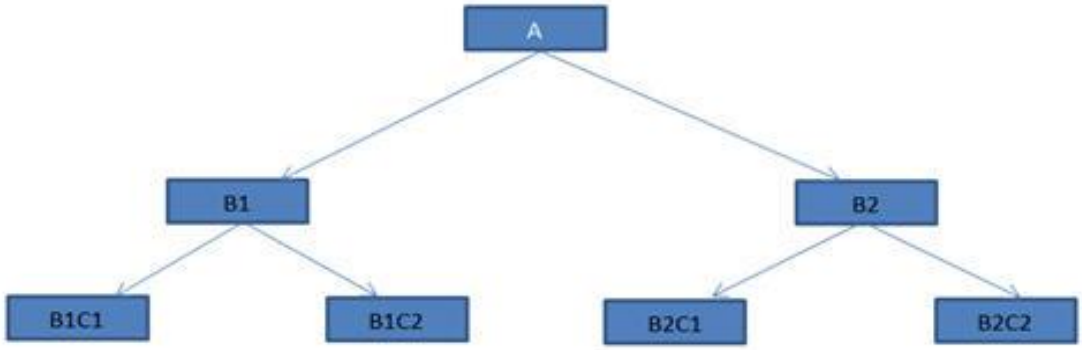


Decomposition Based Integration:

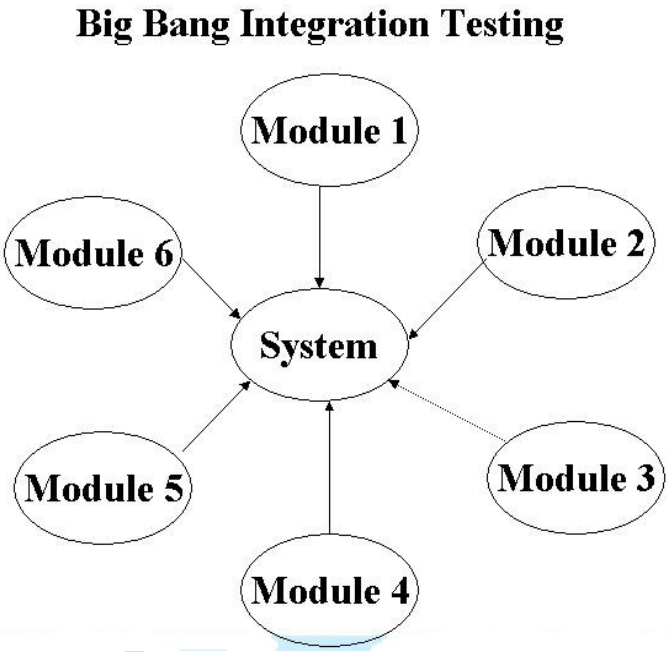
In this strategy, we do the decomposition based on the functional characteristics of the system. A functional characteristic is defined by what the module does, that is, actions or activities performed by the module. In this strategy our main goal is to test the interfaces among separately tested units. There are four approaches for this strategy:

--> Non Incremental Integration Testing / Big bang integration
--> Incremental Integration Testing
--Top-Down Integration
--Bottom-up Integration
-->Practical Approach for Integration Testing / Sandwich integration.

Briefly, big-bang groups the whole system and test it in a single test phase. Top-down starts at the root of the tree and slowly work to lower level of the tree. Bottom-up mirrors top-down, it starts at the lower level implementation of the system and work towards the main. Sandwich is an approach that combines both top-down and bottom-up.



Non Incremental Integration Testing / Big bang integration:
--This is one of the easiest approaches to apply in integration testing.
--Here we treat the whole system as a subsystem and test it in a single test phase.
--Normally this means simply integrating all the modules, compile them all at once and then test the resulting system.
--This approach requires little resources to execute as we do not need to identify critical components (like interactions, paths between the modules) nor require extra coding for the "dummy modules".
--This approach may be used for very small systems, however it is still not recommended because it is not systematic.
--In larger systems, the low resources requirement in executing this testing is easily offset by the resources required to locate the problem when it occurs.



In summary, big bang integration has the following characteristics:
• Considers the whole system as a subsystem
• Tests all the modules in a single test session
• Only one integration testing session

Advantages:
• Low resources requirement
• Does not require extra coding

Disadvantages:
• Not systematic
• Hard to locate problems
• Hard to create test cases

Incremental Integration Testing:
In Incremental integration testing, the developers integrate the modules one by one using stubs or

drivers to uncover the defects. This approach is known as incremental integration testing. To the contrary, big bang is one other integration testing technique, where all the modules are integrated in one shot.

Incremental Integration Testing is beneficial for the following reasons:

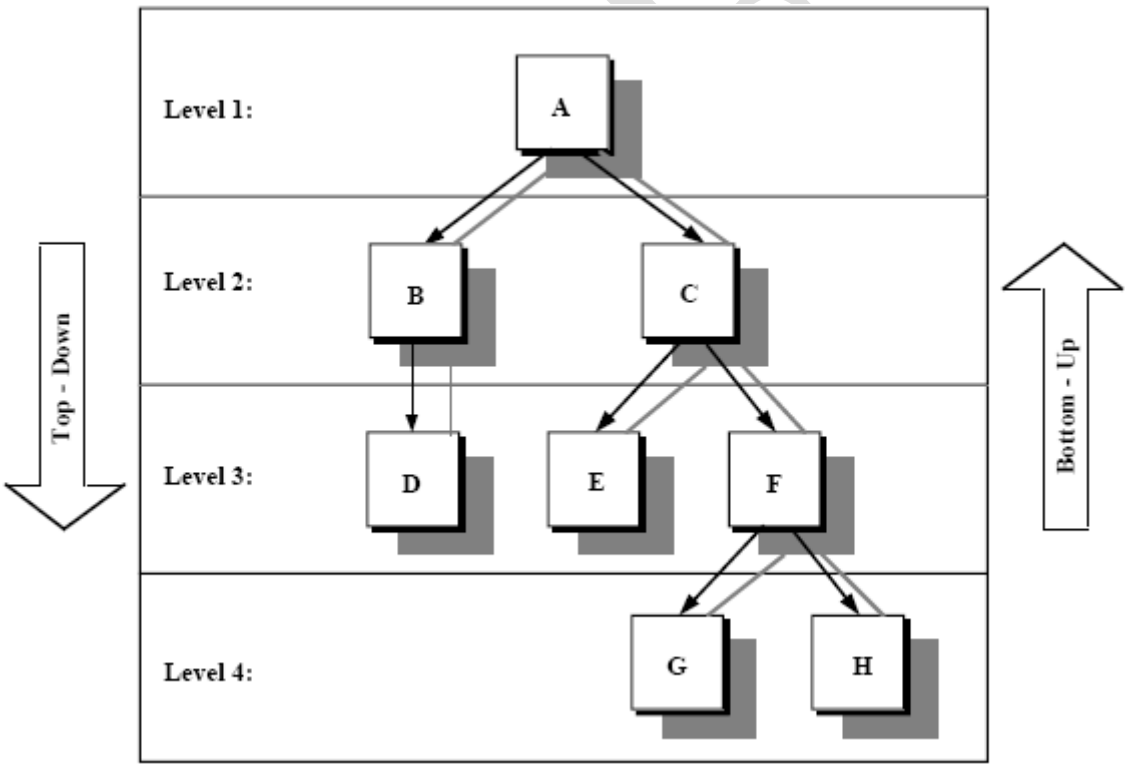
--Each Module provides a definitive role to play in the project/product structure

--Each Module has clearly defined dependencies some of which can be known only at the runtime.

--The incremental integration testing's greater advantage is that the defects are found early in a smaller assembly when it is relatively easy to detect the root cause of the same.

--A disadvantage is that it can be time-consuming since stubs and drivers have to be developed for performing these tests.

Types of Incremental integration testing:
-->Top Down integration testing
--> Bottom-up integration testing



Top Down Integration Testing:
--In top-down integration, we start at the target node at root of the functional decomposition tree and work toward the leaves.
--Stubs are used to replace the children nodes attached to the target node.
--A test phase consists in replacing one of the stub modules with the real code and test the resulting subsystem.
-- If no problem is encountered then we do the next test phase.
--If all the children were replaced by real code at least once and meet the requirements then we

move down to the next level.
--Now we can replace the higher level tested modules with real code and continue the integration testing.
--For top-down integration the number of integration testing sessions is: nodes−leaves +edges.
Top-down integration has the drawback of requiring stubs:
--While stubs are simpler than the real code, it is not straightforward to write them; the emulation must be complete and realistic, that is, the test cases results ran on the stub should match with the results on the real code.
--Being a throw-away code, it does not reach the final product nor it will increase functionality of the software thus it is extra programming effort without a direct reward.

Modules subordinate to the top module are integrated in the following two ways:

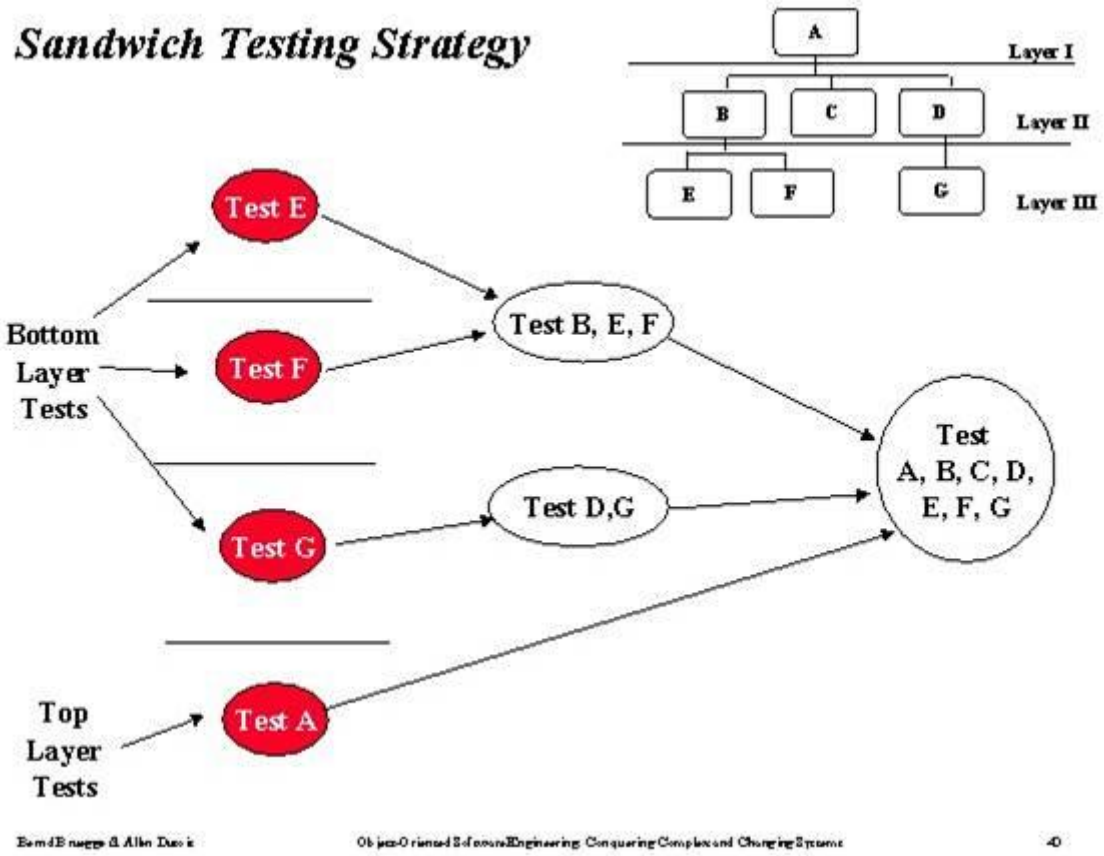
Depth First Integration: In this type, all modules on a major control path of the design hierarchy are integrated first. In the figure shown above, modules A, B, and D are integrated first, next modules A, C, E, F G, h and integrated.

Breadth First Integration: In this type, modules directly subordinate at each level, moving across the design hierarchy horizontally are integrated first. In the figure shown above, modules B,C are integrated first, next modules D, E, F and at last modules G, H integrated.

Bottom-up Integration Testing:
--Bottom-up integration starts at the opposite end of the functional decomposition tree, instead of starting at the main program we start at the lower-level implementation of the software.
--By moving in an opposite direction, the parent nodes are replaced by throw-away codes instead of the children.
--These throw-away codes are also known as drivers.
--This approach allow us to start working with simpler and lower level of the implementation, allowing us to create testing environments more easily because of the simpler outputs of those modules.
--This also allows us to handle the exceptions more easily.
--Conversely, we do not have an early prototype thus the main program is the last to be tested. If there is a design error then it will be just identified at a later stage, which implies high error correction cost.
--Bottom-up integration is commonly used for object-oriented systems, real-time systems and systems with strict performance requirements.
--For bottom-up integration the number of integration testing sessions is: nodes−leaves+edges.

Issue	Top-Down Testing	Bottom-Up Testing
Architectural Design	It discovers errors in high-level design, thus detects errors at an early stage.	High-level design is validated at a later stage.
System Demonstration	Since we integrate the modules from top to bottom, the high-level design slowly expands as a working system. Therefore, feasibility of the system can be demonstrated to the top management.	It may not be possible to show the feasibility of the design. However, if some modules are already built as reusable components, then it may be possible to produce some kind of demonstration.
Test Implementation	*(nodes – 1)* stubs are required for the subordinate modules.	*(nodes – leaves)* test drivers are required for super-ordinate modules to test the lower-level modules.

Practical Approach for Integration Testing / Sandwich integration:
--Sandwich integration combines top-down integration and bottom-up integration.
--The main concept is to maximize the advantages of top-down and bottom-up and minimizing their weaknesses.
--Sandwich integration uses a mixed-up approach where we use stubs at the higher level of the tree and drivers at the lower level (Figure).
-- The testing direction starts from both side of tree and converges to the centre, thus the term sandwich.
--This will allow us to test both the top and bottom layers in parallel and decrease the number of stubs and drivers required in integration testing.

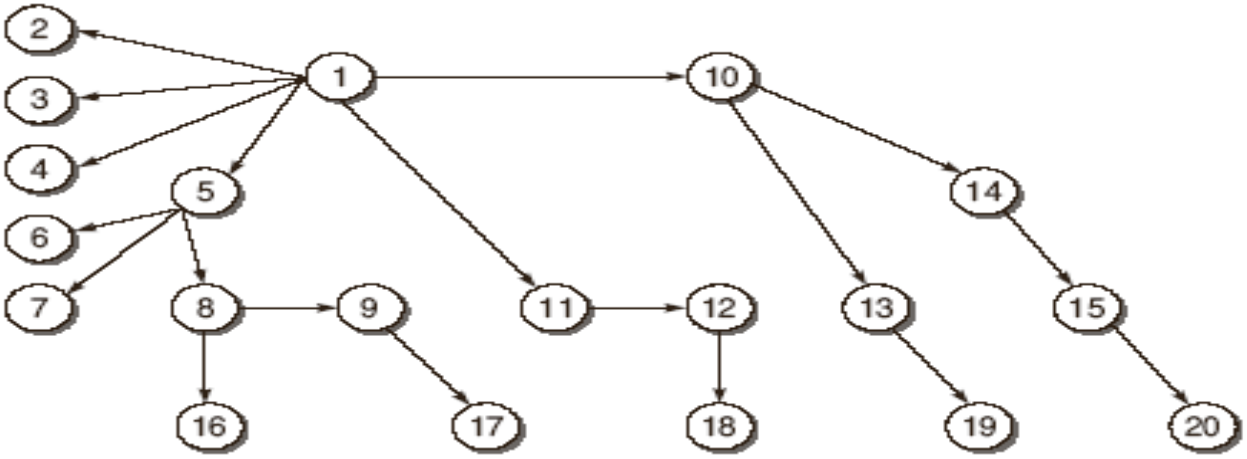


Advantages:
--Top and bottom layers can be done in parallel
-- Less stubs and drivers needed
-- Easy to construct test cases
--Better coverage control
--Integration is done as soon a component is implemented

Disadvantages:
-- Still requires throw-away code programming
-- Partial big bang integration
-- Hard to isolate problems

Call-Graph based Integration:

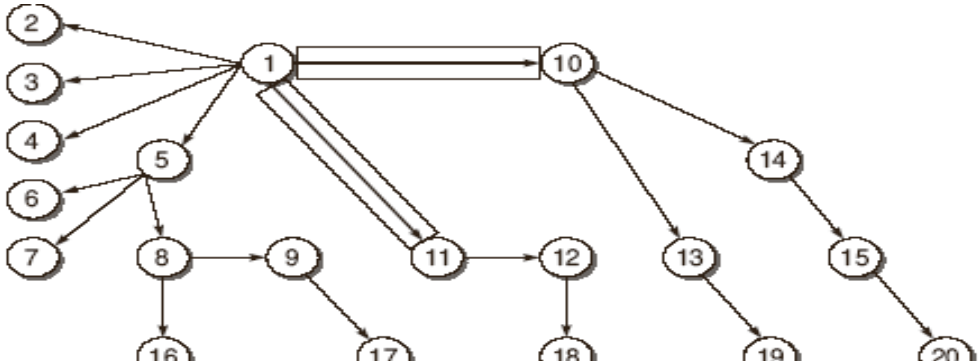
A call graph is a directed graph, where the nodes are either modules or units, and a directed edge from one node to another node means one module has called another module. The call graph can be captured in a matrix form which is known as the adjacency matrix.



1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	×	×	×	×					×	×									
2																			
3																			
4																			
5					×	×	×												
6																			
7																			
8								×							×				
9																×			
10												×	×						
11											×								
12																	×		
13																		×	
14														×					
15																			×
16																			
17																			
18																			
19																			
20																			

There are two types of integration testing based on call graph:
-->Pair wise Integration
-->Neighbourhood Integration.

Pair wise Integration:
--In pair-wise integration, we eliminate the need of stub and driver, by using the real code instead. --This is similar to big bang where has problem isolation problem due to the large amount of modules we are testing at once.
--By pairing up the modules using the edges, we will have a number of test sessions equal to the number of edges that exist in the call graph.
--Since the edges correspond to functions or procedures invocated, in a standard system, this implies many test sessions.
--For pair-wise integration the number of integration testing sessions is the number of edges.



Neighbourhood Integration:
--While pair-wise integration eliminates the need of stub and driver, it still requires many test cases.
--As an attempt of improving from pair-wise, neighbourhood requires fewer test cases.
--In neighbourhood integration, we create a subsystem for a test session by having a target node and grouping all the nodes near it.
--Near is defined as nodes that are linked to the target node that is an immediate predecessor or successor of it.
--By doing this we will be able to reduce considerably the amount of test sessions required.
--The total test sessions in neighbourhood integration can be calculated as:

Neighbourhood = nodes – sink nodes
= 20 - 10
= 10

where Sink Node is an instruction in a module at which execution terminates.

Node	Neighbourhoods	
	Predecessors	Successors
1	-----	2,3,4,5,10,11
5	1	6,7,8
8	5	9,16
9	8	17
10	1	13,14
11	1	12
12	11	18
13	10	19
14	10	15
15	14	20

Path Based Integration:
--By moving to path-based integration we will be approaching integration testing from a new direction. Here we will try to combine both structural and functional approach in path-base integration.
--Finally, instead of testing the interfaces (which are structural), we will be testing the interactions (which are behavioural).
--Here, when a unit is executed certain path of source statements is traversed.
--When this unit calls source statements from another unit, the control is passed from the calling unit to the called unit.
--For integration testing we treat these unit calls as an exit followed by an entry.

We need to understand the following definitions for path-based integration:

Source Node: A program statement fragment at which program execution begins or resumes.

Sink Node: A statement fragment at which program execution terminates

Module execution path (MEP): A sequence of statements within a module that begins with a source node, Ends with a sink node with no intervening sink nodes.

Message: A programming language mechanism by which one unit transfers control to another unit. Usually interpreted as subroutine / function invocations .The unit which receives the message always returns control to the message source.
MM-path: A module to module path. It is an interleaved sequence of module execution paths and messages which are used to describe sequences of module execution paths that include transfers of control among separate units. MM-paths always represent feasible execution paths, and these paths cross unit boundaries.

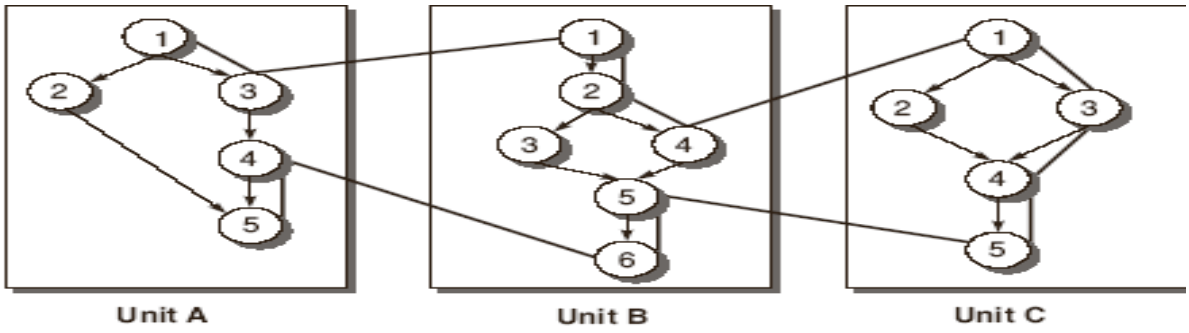


Figure 7.12 MM-path

Table 7.3 MM-path details

	Source Nodes	Sink Nodes	MEPs
Unit A	1,4	3,5	MEP(A,1) = <1,2,5> MEP(A,2) = <1,3> MEP(A,3) = <4,5>
Unit B	1,5	4,6	MEP(B,1) = <1,2,4> MEP(B,2) = <5,6> MEP(B,3) = <1,2,3,4,5,6>
Unit C	1	5	MEP(C,1) = <1,3,4,5> MEP(C,2) = <1,2,4,5>

Function Testing:

Function testing is defined as "the process of attempting to detect discrepancies between the functional specifications of a software and its actual behaviour". When an integrated system is tested, all its specified functions and external interfaces are tested on the software. Every functionality of the system specified in the functions is tested according to its external specifications. The function test must determine if each component or business event:

--Performs in accordance to the specifications
--Responds correctly to all conditions that may be presented by incoming events / data,
--Moves data correctly from one business event to the next (including data stores)
--Business events initiated in the order required to meet the business objectives of the system.

An effective function test cycle must have a defined set of processes and deliverables. The primary processes / deliverables for requirements based function test are:

Test Planning: During planning, the test leader with assistance from the test team defines the scope, schedule, and deliverables for the function test cycle.

Partioning / Functional Decomposition: Functional decomposition of a system is the breakdown of a system into functional components or functional areas.

Requirement Definition: The testing organization needs specified requirements in the form of proper documents to proceed with the function test.

Test case design: A tester designs and implements a test case to validate that the product performs in accordance with the requirements.

Traceability matrix formation: Test cases need to be traced / mapped back to the appropriate requirement. A function coverage matrix is prepared. This matrix is a table, listing specific functions to be tested, the priority for testing each function, and test cases that contain tests for each function.

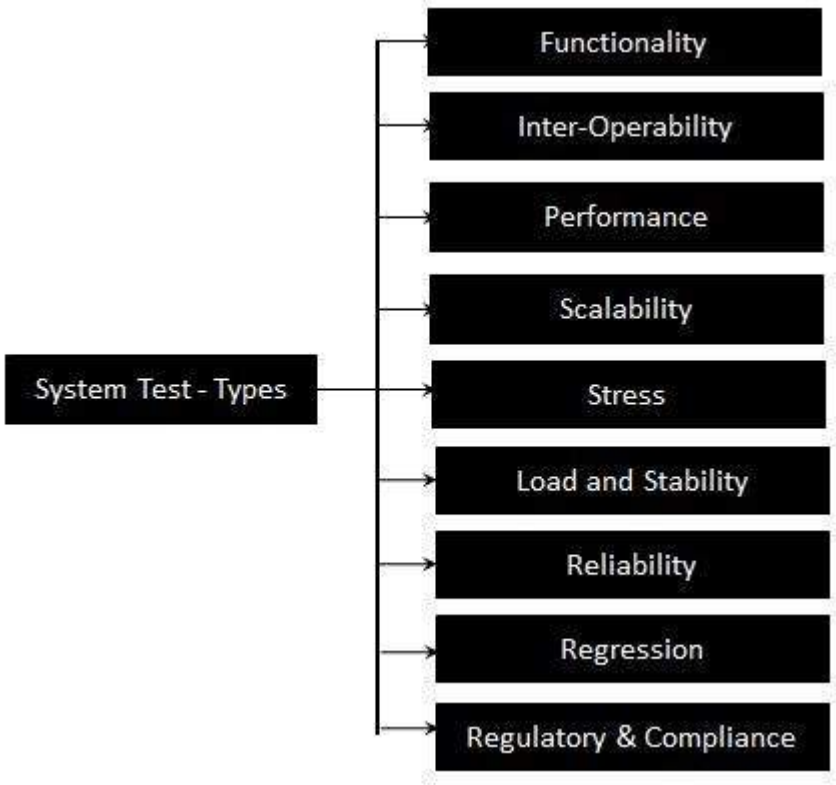
Functions / Features	Priority	Test Cases
F1	3	T2, T4, T6
F2	1	T1. T3, T5

Test case execution: As in all the phases of testing, an appropriate set of test cases need to be executed and the results of those test cases recorded.

System Testing:

--System testing is the type of testing to check the behaviour of a complete and fully integrated software product based on the software requirements specification (SRS) document.
--The main focus of this testing is to evaluate Business / Functional / End-user requirements.
--This is black box type of testing where external working of the software is evaluated with the help of requirement documents & it is totally based on Users point of view.
--For this type of testing do not required knowledge of internal design or structure or code.
--This testing is to be carried out only after System Integration Testing is completed where both Functional & Non-Functional requirements are verified.
--In the integration testing testers are concentrated on finding bugs/defects on integrated modules. But in the Software System Testing testers are concentrated on finding bugs/defects based on software application behaviour, software design and expectation of end user.

Categories of System Testing:



Recovery Testing:
Recovery is just like the exception handling feature of a programming language. It is a type of non-functional testing. Recovery testing is done in order to check how fast and better the application can recover after it has gone through any type of crash or hardware failure etc. Recovery testing is the forced failure of the software in a variety of ways to verify that recovery is properly performed. Thus Recovery Testing is " the activity of testing how well the software is able to recover from crashes, hardware failures, and other similar problems".

Some examples of Recovery testing are:
--When an application is receiving data from a network, unplug the connecting cable. After some time, plug the cable back in and analyze the application's ability to continue receiving data from the point at which the network connection was broken.
--Restart the system while a browser has a definite number of sessions and check whether the browser is able to recover all of them or not.

"Biezer" proposes that testers should work on the following areas during recovery testing:
Restart: Testers must ensure that al transactions have been reconstructed correctly and that all devices are in proper states.

Switchover: Recovery can also be done if there are standby components and in case of failure of one component, the standby takes over the control.

Security Testing:
--It is a type of non-functional testing.
--Security testing is basically a type of software testing that's done to check whether the application or the product is secured or not.
--It checks to see if the application is vulnerable to attacks, if anyone hack the system or login to the application without any authorization.
--It is a process to determine that an information system protects data and maintains functionality as intended.
--The security testing is performed to check whether there is any information leakage in the sense by encrypting the application or using wide range of software's and hardware's and firewall etc.
--Software security is about making software behave in the presence of a malicious attack.

Types of Security Requirements:
-- Security Requirements should be associated with each functional requirement.
--In addition to security concerns that are directly related to particular requirements, a software project has security issues that are global in nature.

How to perform security testing:
Testers must use a risk based approach, grounded in both the systems architectural reality and the attackers mindset, to gauge software security adequately. By identifying risks and potential loss associated with those risks in the system and creating tests driven by those risks, the tester can properly focus on areas of code in which an attack is likely to succeed.

Elements of Security Testing:
--Confidentiality
--Integrity

--Authentication
--Availability
--Authorization
--Non-repudiation.

Performance Testing:

--Software performance testing is a means of quality assurance (QA).
--It involves testing software applications to ensure they will perform well under their expected workload.
--Features and Functionality supported by a software system is not the only concern. A software application's performance like its response time, do matter.
--The goal of performance testing is not to find bugs but to eliminate performance bottlenecks
--Performance testing is done to provide stakeholders with information about their application regarding speed, stability and scalability.
--More importantly, performance testing uncovers what needs to be improved before the product goes to market.
-- Without performance testing, software is likely to suffer from issues such as: running slow while several users use it simultaneously, inconsistencies across different operating systems and poor usability.
--Performance testing will determine whether or not their software meets speed, scalability and stability requirements under expected workloads.
--Applications sent to market with poor performance metrics due to non existent or poor performance testing are likely to gain a bad reputation and fail to meet expected sales goals.
--Also, mission critical applications like space launch programs or life saving medical equipments should be performance tested to ensure that they run for a long period of time without deviations.

The following tasks must be done for this thing:
-- Develop high level plan including requirements, resources, timeliness, and milestones.
-- Develop a detailed performance test plan.
-- Specify test data needed.
-- Execute tests probably repeatedly on order to se whether any unaccounted factor might affect the results.

Load Testing:

--Load testing is a type of non-functional testing.
--A load test is type of software testing which is conducted to understand the behaviour of the application under a specific expected load.
--Load testing is performed to determine a system’s behaviour under both normal and at peak conditions.
--It helps to identify the maximum operating capacity of an application as well as any bottlenecks and determine which element is causing degradation. E.g. If the number of users are increased then how much CPU, memory will be consumed, what is the network and bandwidth response time.
--Load testing can be done under controlled lab conditions to compare the capabilities of different systems or to accurately measure the capabilities of a single system.
--Load testing involves simulating real-life user load for the target application. It helps you determine how your application behaves when multiple users hits it simultaneously.
•Load testing differs from stress testing, which evaluates the extent to which a system keeps working when subjected to extreme work loads or when some of its hardware or software has been compromised.
--The primary goal of load testing is to define the maximum amount of work a system can handle without significant performance degradation.
Examples of load testing include:
•Downloading a series of large files from the internet.

- Running multiple applications on a computer or server simultaneously.
- Assigning many jobs to a printer in a queue.
- Subjecting a server to a large amount of traffic.
- Writing and reading data to and from a hard disk continuously.

Stress Testing:

- It is a type of non-functional testing.
- It involves testing beyond normal operational capacity, often to a breaking point, in order to observe the results.
- It is a form of software testing that is used to determine the stability of a given system.
- It puts greater emphasis on robustness, availability, and error handling under a heavy load, rather than on what would be considered correct behaviour under normal circumstances.
- The goals of such tests may be to ensure the software does not crash in conditions of insufficient computational resources (such as memory or disk space).
- Thus "*Stress Testing tries to break the system under test by overwhelming its resources in order to find the circumstances under which it will crash*"
- The areas that may be stressed in a system are: Input Transactions, Disk Space, Output, Communications, Interaction with users.

Usability Testing:

--Usability testing is an essential element of quality assurance.
-- It is the measure of a product's potential to accomplish the goals of the user.
--Usability testing is a method by which users of a product are asked to perform certain tasks in an effort to measure the product's ease-of-use, task time, and the user's perception of the experience. -- This look as a unique usability practice because it provides direct input on how real users use the system.
--Usability testing measures human-usable products to fulfil the users purpose.
--The item which takes benefit from usability testing are web sites or web applications, documents, computer interfaces, consumer products, and devices.
--Usability testing processes the usability of a particular object or group of objects, where common human-computer interaction studies try to formulate universal principles.

What the user wants or excepts from the system can be determined using several ways like:
--Area Experts,
--Group meetings
--Surveys
--Analyse similar products

Usability characteristics against which testing is conducted are:
--Ease of Use
--Interface steps
--Response Time
--Help System
--Error Messages

Compatability / Conversion/Configuration Testing:

--Compatibility is a non- functional testing to ensure customer satisfaction.
-- It is to determine whether your software application or product is proficient enough to run in different browsers, database, hardware, operating system, mobile devices and networks.
--Application could also impact due to different versions, resolution, internet speed and configuration etc. Hence it's important to test the application in all possible manners to reduce failures and overcome from embarrassments of bug's leakage.
-- As a Non- functional tests, Compatibility testing is to endorse that the application runs properly in

different browsers, versions, OS and networks successfully.
--Compatibility test should always perform on real environment instead of virtual environment.
Test the compatibility of application with different browsers and operating systems to guarantee 100% coverage.
Types of Software compatibility testing:
- Browser compatibility testing
- Hardware
- Networks
- Mobile Devices
- Operating System
- Versions

Acceptance Testing:

--After the system test has corrected all or most defects, the system will be delivered to the user or customer for acceptance testing.
--Acceptance testing is basically done by the user or customer although other stakeholders may be involved as well.
--The goal of acceptance testing is to establish confidence in the system.
--Acceptance testing is most often focused on a validation type testing.
--Thus "*Acceptance Testing is the formal testing conducted to determine whether a software system satisfies its acceptance criteria and to enable buyer to determine whether to accept the system or not."*
--Thus acceptance testing is designed to:

-Determine whether the software is fit for the user to use.
-Making users confident about product
-Determine whether a software system satisfies its acceptance criteria.
-Enable the buyer to determine whether to accept the system.

Types of Acceptance Testing:
--> Alpha Testing
--> Beta Testing

Alpha Testing:

--Alpha testing is one of the most common software testing strategy used in software development. Its specially used by product development organizations.
--This test takes place at the developer's site. Developers observe the users and note problems.
--Alpha testing is testing of an application when development is about to complete. Minor design changes can still be made as a result of alpha testing.
--Alpha testing is typically performed by a group that is independent of the design team, but still within the company, e.g. in-house software test engineers, or software QA engineers.
--Alpha testing is final testing before the software is released to the general public. **It has two phases:**
-->Alpha testing is simulated or actual operational testing by potential users/customers or an independent test team at the developers' site.
-->Alpha testing is often employed for off-the-shelf software as a form of internal acceptance testing, before the software goes to beta testing.

Entry Criteria for Alpha:
--All features are complete / testable
--High bugs on primary platform are fixed / verified.
--50% of medium bugs on primary platforms are fixed / verified.
--All features are tested on the primary platforms.

--Performance has been measured / compared
--Alpha sites are ready for installation.

Exit criteria to Alpha:
-- Get response / feedbacks from the customers.
--Prepare a report of any serious bugs being noticed.
--Notify bug – fixing issues to developers.

Beta Testing:
--In software development, a beta test is the second phase of software testing in which a sampling of the intended audience tries the product out.
--It is also known as field testing. It takes place at customer's site. It sends the system to users who install it and use it under real-world working conditions.
--Beta is the second letter of the Greek alphabet.
--Originally, the term *alpha test* meant the first phase of testing in a software development process. The first phase includes unit testing, component testing, and system testing.
--Beta testing can be considered "pre-release testing."
--Beta testing is also sometimes referred to as user acceptance testing (UAT) or end user testing.
--In this phase of software development, applications are subjected to real world testing by the intended audience for the software.
--The experiences of the early users are forwarded back to the developers who make final changes before releasing the software commercially.

Entry Criteria for Beta:
--Positive responses from alpha site.
--Customer bugs in alpha testing have been addressed.
--There are no fatal errors which can affect the functionality of the software.
--Beta sites are ready for installation.

Exit criteria to Beta:
-- Get response / feedbacks from the beta testers.
--Prepare a report of all serious bugs.
--Notify bug–fixing issues to developers.

Regression testing

Progressive Vs regressive testing, Regression testability, Objectives of regression testing, When regression testing done?, Regression testing types, Regression testing techniques

Progressive Vs Regressive testing:
--All the test case design methods or testing technique, discussed till now are referred to as progressive testing or development testing.
--The purpose of regression testing is to confirm that a recent program or code change has not adversely affected existing features.
--Regression testing is nothing but full or partial selection of already executed test cases which are re-executed to ensure existing functionalities work fine.
--This testing is done to make sure that new code changes should not have side effects on the existing functionalities.
--It ensures that old code still works once the new code changes are done.

Need of Regression Testing:
- Change in requirements and code is modified according to the requirement

- New feature is added to the software
- Defect fixing
- Performance issue fix

Definition:

Regression testing is the selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements.

Regression Testability:

Regression testability refers to the property of a program, modification or test suite that lets it be effectively and efficiently regression-tested. We can classify a program as regression testable if most single statement modifications to the program entail(involves) rerunning a small proportion of the current test suite.

Objectives of Regression Testing:

--***It tests to check that the bug has been addressed:*** The first objective in bug fix testing is to check whether the bug-fixing has worked or not.

--***It finds other related bugs:*** Regression tests are necessary to validate that the system does not have any related bugs.

--***It tests to check the effect on other parts of the program****:* It may be possible that bug-fixing has unwanted consequences on other parts of a program. Therefore, it is necessary to check the influence of changes in one part or other parts of the program.

When regression testing done?

Software Maintenance:

--*Corrective Maintenance***:** Changes made to correct a system after a failure has been observed.

--*Adaptive Maintenance:* Changes made to achieve continuing compatibility with the target environemnt or other systems.

--*Perfective Maintenance:* Changes made to improve or add capabilities.

--*Preventive Maintenance:* Changes made to increase robustness, maintainability, portability, and other features.

Rapid Iterative Development: The extreme programming approach requires that a test be developed for each class and that this test be re-run every time the class changes.

Compatability Assesment and Benchmarking: Some test suites designed to be run on a wide range of platforms and applications to establish conformance with a standard or to evaluate time and space performance.

Regression Testing Types:

Bug-fix Regression: This testing is performed after a bug has been reported and fixed.

Side-Effect Regression / Stability Regression: It involves retesting a substantial part of the product. The goal is to prove that the changes has no detrimental effect on something that was earlier in order.

Regression Testing Techniques:

There are different techniques for regression testing. They are:

-->**Regression test selection technique**: This technique attempts to reduce the time required to retest a modified program by selecting some subset of the existing test suite.

-->**Testcase prioritization technique:** Regression test prioritization attempts to reorder a regression

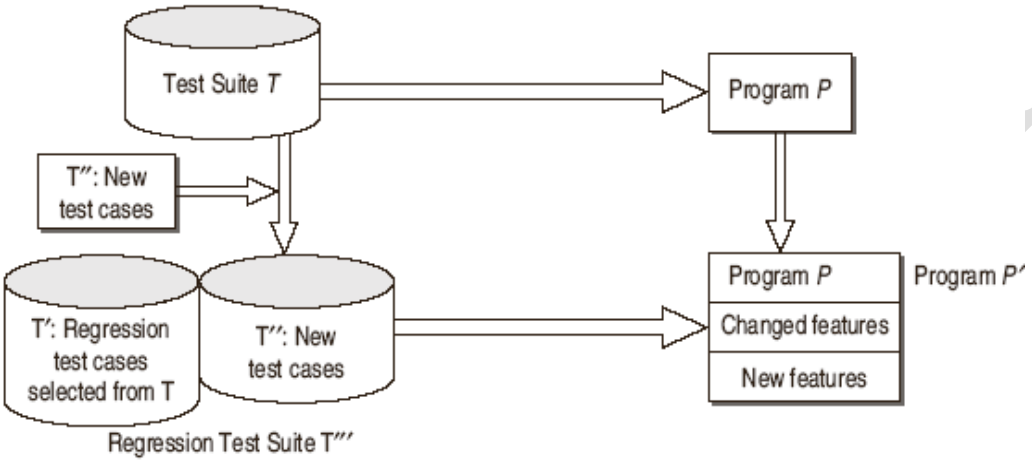
test suite so that those tests with the highest priority according to some established criteria, are executed earlier in the regression testing process rather than those lower priority. There are two types of prioritization:

(a) General Test Case Prioritization: For a given program P and test suits T, we prioritize the test cases in T that will be useful over a succession of subsequent modified versions of P, without any knowledge of the modified version.

(b) Version-Specific Test case Prioritization: We prioritize the test cases in T, when P is modified to P', with the knowledgeof the changes made in P.

-->**Test Suite Reduction Technique:** It reduces testing costs by permanently eliminating redundant test cases form test suites in terms of codes of functionalities exercised.

Selective Retest Technique:



Selective retest technique attempts to reduce the cost of testing by identifying the portions of P' (modified version of Program) that must be exercised but the regression test suite. Following are the characteristic features of the selective retest technique:
--> It minimizes the resources required to regression test a new version.
--> It is achieved by minimizing the number of test cases applied to the new version.
--> It analyses the relationship between the test cases and the software elements they cover.
-->It uses the information about changes to select test cases.

Steps in Selective retest technique:
1. Select T' subset of T, a set of test cases to execute on P'.
2. Test P' with T', establishing correctness of P' with respect to T'.
3. If necessary, create T'", a set of new functional test cases for P'.
4. Test P' with T', establishing correctness of P' with respect to T".
5. Create T"'. a new test suite and test execution profile for P', from T, T' and T".

Strategy for Test Case Selection:

For large software systems, there may be thousands of test cases available in its test suite. When a change is introduced into the system for next version, rerunning all the test cases is a costly and time consuming task. Therefore a need for selecting a subset of test cases from the original test suite is necessary. But the use of multiple criteria should increase the code coverage. So, an effective test case selection strategy is to be designed based on the code coverage.

Selection criteria based on Code:
-->Fault revealing test cases
-->Modification revealing Test cases.
-->Modification traversing Test cases.

Regression Test Selection Techniques:

Minimization Techniques: Minimization-based regression test selection techniques attempt to select minimal sets of test cases from T that yield coverage of modified or affected portions of P. For example, this technique uses systems of linear equations to express relationships between test cases and basic blocks (single-entry, single-exit sequences of statements in a procedure). The technique uses a 0-1 integer programming algorithm to identify a subset T' of T that ensures that every segment that is statically reachable from a modified segment is exercised by at least one test case in T9 that also exercises the modified segment.

Dataflow Techniques: Dataflow-coverage-based regression test selection techniques select test cases that exercise data interactions that have been affected by modifications. For example, the technique requires that every definition-use pair that is deleted from P, new in P', or modified for P' be tested. The technique selects every test case in T that, when executed on P, exercised deleted or modified definition-use pairs, or executed a statement containing a modified predicate.

Safe Techniques: Most regression test selection techniques—minimization and dataflow techniques among them—are not designed to be safe. Techniques that are not safe can fail to select a test case that would have revealed a fault in the modified program. In contrast, when an explicit set of safety conditions can be satisfied, safe regression test selection techniques guarantee that the selected subset, T', contains all test cases in the original test suite T that can reveal faults in P'

Ad Hoc/Random Techniques: When time constraints prohibit the use of a retest-all approach, but no test selection tool is available, developers often select test cases based on "hunches," or loose associations of test cases with functionality. Another simple approach is to randomly select a predetermined number of test cases from T.

Retest-All Technique: The retest-all technique simply reuses all existing test cases. To test P', the technique effectively "selects" all test cases in T.

Evaluating Regression Test Selection Technique:

Inclusiveness: Let M be a regression test selection technique. Inclusiveness measures the extent to which M chooses modification revealing tests from T for inclusion in T'. We define inclusiveness relative to a particular program, modified program, and test suite, as follows:

DEFINITION
Suppose T contains n tests that are modification revealing for P and P', and suppose M selects m of these tests. The inclusiveness of M relative to P, P', and T is
1) the percentage given by the expression (100(m/n))
if n# 0 or 2) 100% if n = 0.
For example, if T contains 50 tests of which eight are modification-revealing for P and P', and M selects two of these eight tests, then M is 25% inclusive relative to P, P', and T. If T contains no modification-revealing tests then every test selection technique is 100% inclusive relative to P, P'', and T.

Precision: Let M be a regression test selection technique. Precision measures the extent to which M omits tests that are non modification-revealing. We define precision relative to a particular program,

modified program, and test suite, as follows:

DEFINITION
Suppose T contains n tests that are non modification-revealing for P and P' and suppose M omits m of these tests. The precision of M relative to P, P: and T is
1) the percentage given by the expression (100(m/n)) if
2) 100% if n = 0. n#0,or
For example, if T contains 50 tests of which 44 are non modification-revealing for P and P', and M omits 33 of these 44 tests, then M is 75% precise relative to P, P', and T. If T contains no non-modification-revealing tests, then every test selection technique is 100% precise relative to P, P', and T.

Efficiency: We measure the efficiency of regression test selection techniques in terms of their space and time requirements. Where time is concerned, a test selection technique is more economical than the retest-all technique if the cost of selecting T' is less than the cost of running the tests in T-T' Space efficiency primarily depends on the test history and program analysis information a technique must store. Thus, both space and time efficiency depend on the size of the test suite that a technique selects, and on the computational cost of that technique.

Regression Test Prioritization:

The regression test prioritization approach is different as compared to selective retest techniques. Regression test prioritization attempts to reorder a regression test suite so that those tests with the highest priority, according to some established criterion, are executed earlier in the regression testing process than those with a lower priority.

The steps for this approach are:
1. Select T ' from T, a set of test cases to execute on P'.
2. Produce T '$_P$, a permutation of T ', such that T '$_P$ will have a better rate of fault detection than T '.
3. Test P' with T '$_P$ in order to establish the correctness of P' wrt T '$_P$.
4. If necessary create T '', a set of new functional or structural tests for P'.
5. Test P' with T '' in order to establish the correctness of P' wrt T ''.
6. Create T ''', a new test suite for P', from T, T '$_P$ and T''.