Validation Activities

Validation is the next step after verification. As shown in Figure, every validation testing focuses on a particular SDLC phase and thereby focuses on a particular class of errors. For example, the purpose of unit validation testing is to find discrepancies between the developed module's functionality and its requirements and interfaces specified in the SRS. Similarly, the purpose of system validation testing is to explore whether the product is consistent with its original objectives. The advantage of this structure of validation testing is that it avoids redundant testing and prevents one from overlooking large classes of errors. Software validation is achieved through a series of black-box tests that demonstrate conformity with requirements. A test plan outlines the classes of tests to be conducted and a test procedure defines specific test cases that will be used to demonstrate conformity with requirements. Both the plan and procedure are designed to ensure that all functional requirements are satisfied, all behavioral characteristics are achieved, all performance requirements are attained, documentation is correct and human-engineered, and other requirements are met.

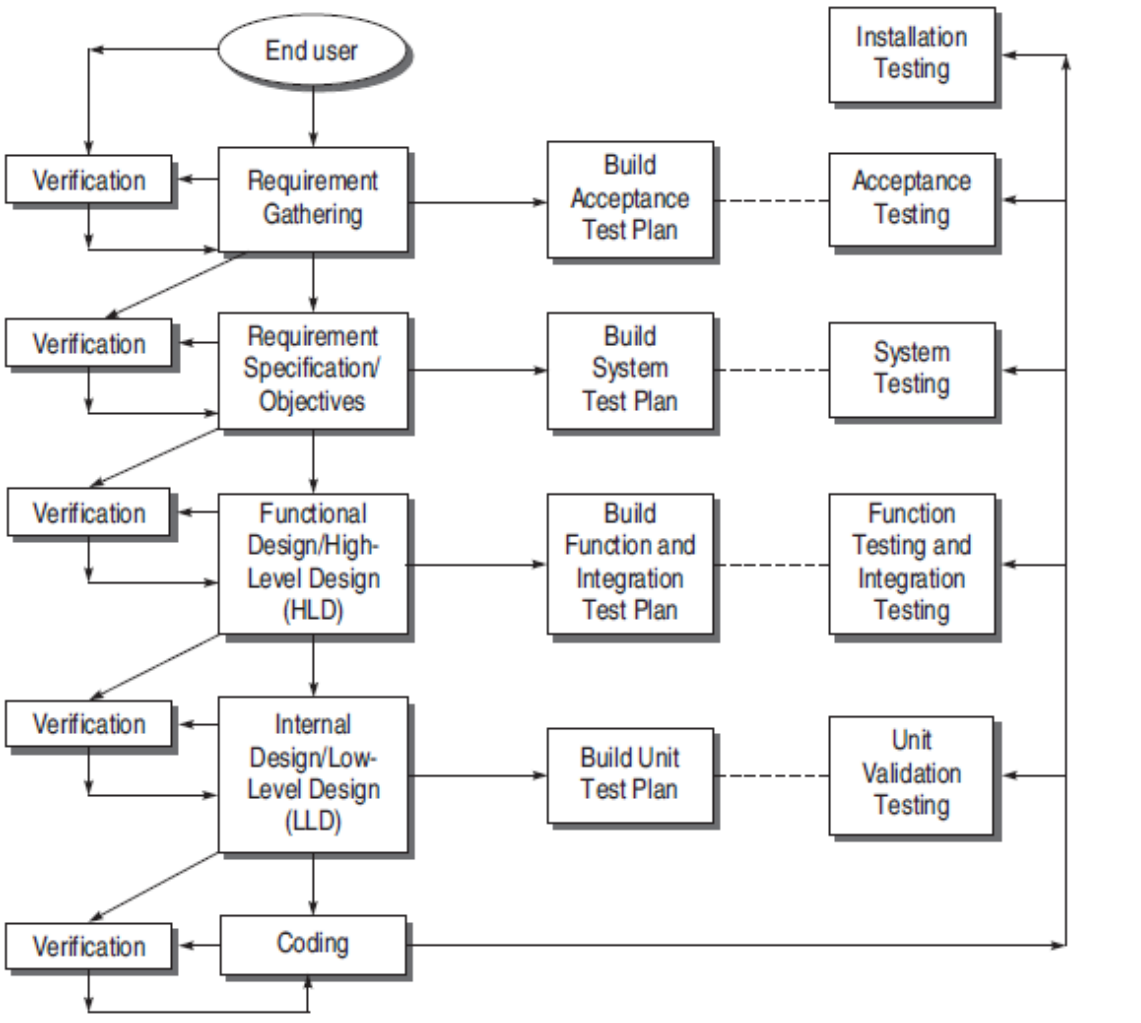


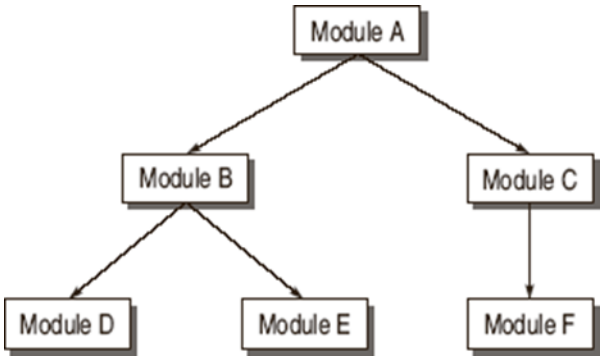
Figure: **V&V activities**

7.1 UNIT VALIDATION TESTING

- Unit is the smallest building block of the software system. Unit testing is normally considered an adjunct to the coding step.
- Units must also be validated to ensure that every unit of software has been built in the right manner in conformance with user requirements.
- Unit tests ensure that the software meets at least a baseline level of functionality prior to integration and system testing.
- A module is not an isolated entity. The module under consideration might be getting some inputs from another module or the module is calling some other module.
- While testing the module, all its interfaces must be simulated if the interfaced modules are not ready at the time of testing the module under consideration.

Drivers

- A test driver can be defined as a software module which is used to invoke a module under test and provide test inputs, control and monitor execution, and report test results or most simplistically, a line of code that calls a method and passes a value to that method.
- Suppose a module is to be tested, wherein some inputs are to be received from another module.
- The module which passes inputs to the module to be tested is not ready and under development then simulate the inputs required in the module to be tested.
- Code is prepared, wherein the required inputs are either hard-coded or entered by the user and passed on to the module under test.
- *This module where the required inputs for the module under test are simulated for the purpose of module or unit testing is known as a driver module*.
- For example, module B /C is under test and module A is not ready which passes inputs to B and C. Therefore, a driver module is needed which will simulate module A in the sense that it passes the required inputs to module B /C as shown in figure.



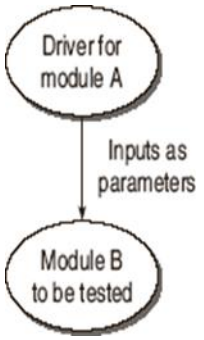


Figure Design hierarchy of an example system Driver Module for module A

A test driver may take inputs in the following form and call the unit to be tested:

- It may hard-code the inputs as parameters of the calling unit.
- It may take the inputs from the user.
- It may read the inputs from a file.

Stubs

- A stub can be defined as a piece of software that works similar to a unit which is referenced by the unit being tested, but it is much simpler than the actual unit.
- A stub works as a 'stand-in' for the subordinate unit and provides the minimum required behaviour for that unit.

- The module under testing may also call some other module which is not ready at the time of testing dummy modules instead of actual modules, which are not ready, are prepared for these subordinate modules. These dummy modules are called *stubs*.

- Module B under test needs to call module D and module E. But they are not ready. Therefore, stubs are designed for module D and module E.

Stubs have the following characteristics:

- Stub is a place holder for the actual module to be called. Therefore, it is not designed with the functionalities performed by the actual module. It is a reduced implementation of the actual module.
- It does not perform any action of its own and returns to the calling unit
- We may include a display instruction as a trace message in the body of stub. The idea is that the module to be called is working fi ne by accepting the input parameters.
- A constant or null must be returned from the body of stub to the calling module.
- Stub may simulate exceptions or abnormal conditions, if required.

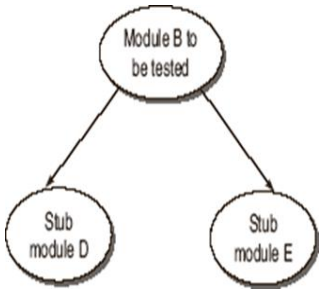


Figure: Stubs

Benefits of Designing Stubs and Drivers

The benefits of designing stubs and drivers are:

- 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 time by 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.

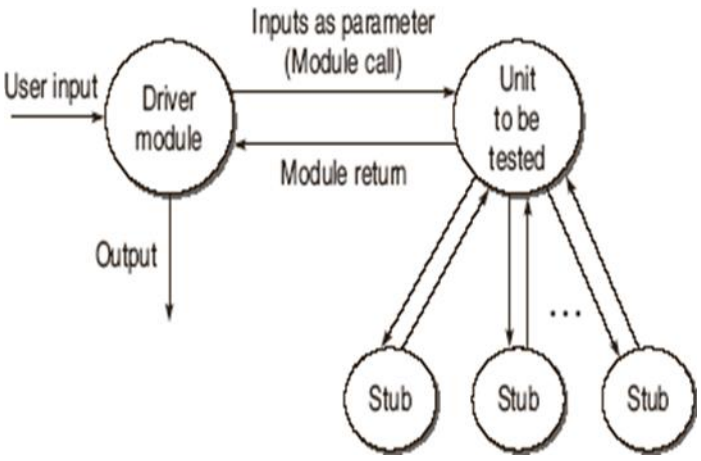


Figure 7.5 Drivers and Stubs

Disadvantages: Drivers and stubs represent overheads also. Overhead of designing them may increase the time and cost of the entire software system. They must be designed simple to keep overheads low. Stubs and drivers are generally prepared by the developer of the module under testing.

Example 7.1

Consider the following program:

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

(a) Suppose main() module is not ready for the testing of calsum() module. Design a driver module for main().

(b) Modules caldiff() and calmul() are not ready when called in main(). Design stubs for these two modules.

Solution

(a) **Driver for** main() **module:**

```
main()
{
        int a, b, c, sum;
        scanf("%d %d %d", &a, &b, &c);
        sum = calsum(a,b,c);
        printf("The output from calsum module is %d", sum);
}
```

(b) **Stub for** caldiff() **Module**

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

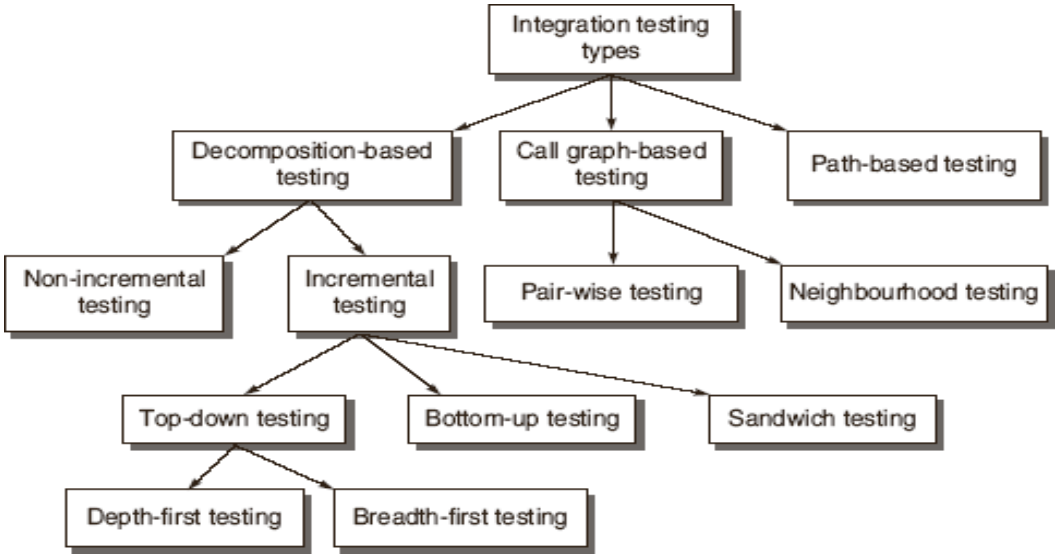
Stub for calmul() **Module**

```
calmul(int x, int y, int z)
{
        printf("Multiplication calculation module");
        return 0;
}
```

7.2 INTEGRATION TESTING

Integrate/combine the unit tested modules one by one and test the behaviour as a combined unit. Integration testing focuses on bugs caused by interfacing between the modules while integrating them.

Various types of integration testing can be seen in a hierarchical tree given below



Integration testing is necessary for the following reasons:

- Integration testing 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.

7.2.1 Decomposition-Based Integration

- This is based on the decomposition of design into functional components or modules.
- In the tree designed for decomposition-based integration, the nodes represent the modules present in the system and the links/edges between the two modules represent the calling sequence.
- The nodes on the last level in the tree are *leaf nodes.*

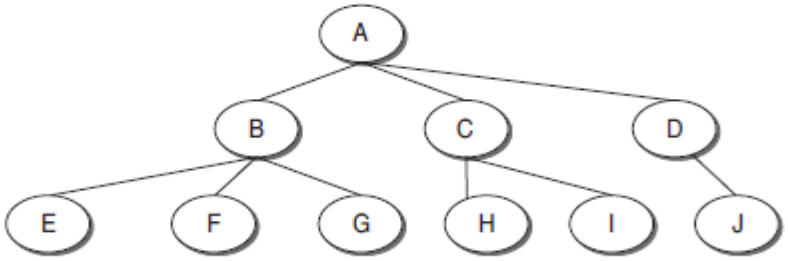


Figure 7.7 Decomposition Tree

Integration method depends on the methods on which the activity of integration is based

(a) To integrate all the modules together and then test it - *non-incremental*
(b) To integrate the modules one by one and test them incrementally - *incremental*

Non-Incremental/Big-Bang Integration Testing

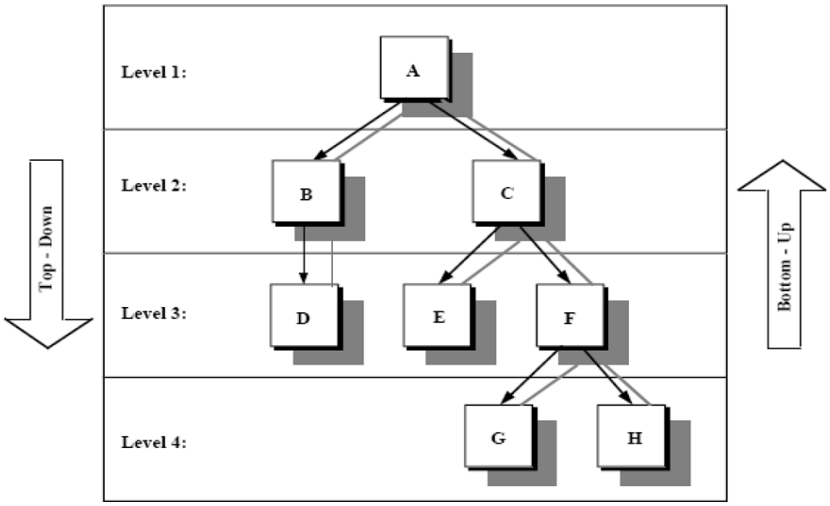
- In this type of testing, either all untested modules are combined together and then tested or unit tested modules are combined together.
- It is also known as Big-Bang integration testing.
- Big-Bang method cannot be adopted practically. This theory has been discarded due to the following reasons:
    - Big-Bang requires more work.
    - Actual modules are not interfaced directly until the end of the software system.
    - It will be difficult to localize the errors since the exact location of bugs cannot be found easily.

Incremental Integration Testing

- In this type, you start with one module and unit test it. Then combine the module which has to be merged with it and perform test on both the modules.
- In this way, incrementally keep on adding the modules and test the recent environment. Thus, an integrated tested software system is achieved.
- Incremental integration testing is beneficial for the following reasons:
  1. Incremental approach does not require many drivers and stubs.
  2. Interfacing errors are uncovered earlier.
  3. It is easy to localize the errors since modules are combined one by one. The first suspect is the recently added module. Thus, debugging becomes easy.
  4. Incremental testing is a more thorough testing.
- It suffers from the problem of serially combining the methods according to the design. But practically, sometimes it is not feasible in the sense that all modules are not ready at the same time.
- According to the big-bang method, all modules should be unit tested independently as they are developed. In this way, there is parallelism.
- As soon as one module is ready, it can be combined and tested again in the integrated environment according to the incremental integration testing.

Types of Incremental Integration Testing

- Incremental integration can be done either from top to bottom or bottom to top. Incremental integration testing is divided into two categories.
  1. Top – down integration Testing
  2. Bottom-up integration testing



Top-down Integration Testing

- Start with the high-level modules and move downward through the design hierarchy.
- 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 example shown in Fig. 7.8, modules 1, 2, 6, 7/8 will be integrated first. Next, modules 1, 3, 4/5 will be integrated.

Breadth first integration

- In this type, all modules directly subordinate at each level, moving across the design hierarchy horizontally, are integrated first.
- In the example shown in Fig. 7.8, modules 2 and 3 will be integrated first. Next, modules 6, 4, and 5 will be integrated. Modules 7 and 8 will be integrated last.

Guidelines can be considered are:

1. In practice, the availability of modules matter the most.
2. If there are critical sections of the software test them as early as possible.
3. I/O modules are added as early as possible so that all interface errors will be detected earlier.

Top-Down Integration Procedure

The procedure for top-down integration process is discussed in the following steps:
1. Start with the top or initial module in the software. Substitute the stubs for all the subordinate modules of top module. Test the top module.
2. After testing the top module, stubs are replaced one at a time with the actual modules for integration.
3. Perform testing on this recent integrated environment.
4. Regression testing may be conducted to ensure that new errors have not appeared.
5. Repeat steps 2–4 for the whole design hierarchy

Drawbacks of top-down integration testing

1. Stubs must be prepared as required for testing one module.
2. Stubs are often more complicated than they first appear.
3. Before the I/O functions are added, the representation of test cases in stubs can be difficult.

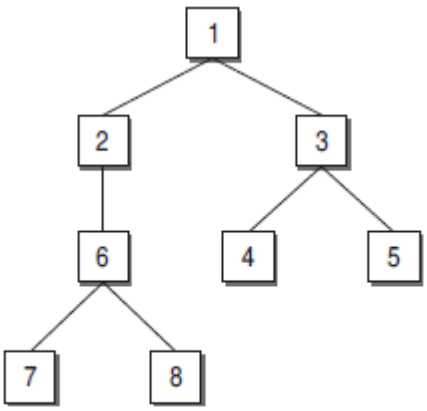
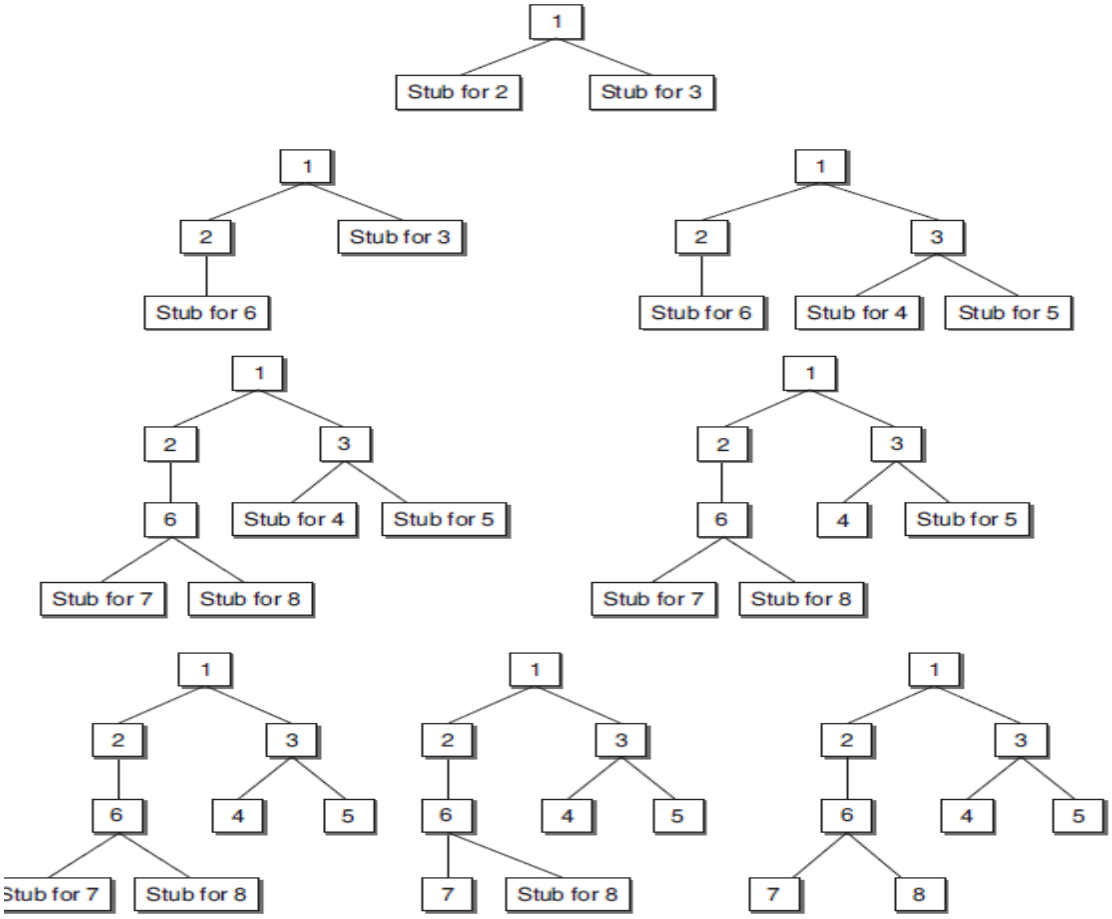
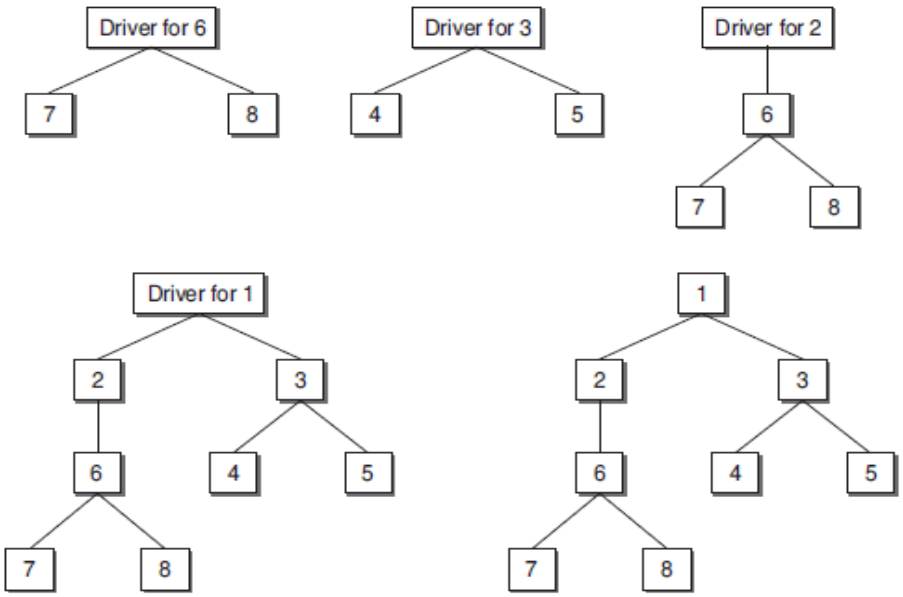


Figure 7.8 Integration Testing

The top-down integration for Fig. 7.8 is shown below.



Bottom-up integration for Fig. 7.8, is shown below:



Bottom-up Integration Testing

- The bottom-up strategy begins with the terminal modules at the lowest level in the software structure.
- After testing these modules, they are integrated and tested moving from bottom to top level.
- Bottom-up integration can be performed at an early stage in the developmental process.
- It is useful for integrating object-oriented systems, real-time systems, and systems with strict performance requirements.
- Bottom-up integration has the disadvantage that the software as a whole does not exist until the last module is added.
- It is not an optimal strategy for functionally decomposed systems, as it tests the most important subsystem last.
- The steps in bottom-up integration are as follows:
    - Start with the lowest level modules in the design hierarchy.
    - Look for the super-ordinate module which calls the module selected in step 1.
    - Test the module selected in step 1 with the driver designed in step 2.
    - The next module to be tested is any module whose subordinate modules have all been tested.
    - Repeat steps 2 to 5 and move up in the design hierarchy.
    - Whenever, the actual modules are available, replace stubs and drivers with the actual one and test again.

Comparison between Top-Down and Bottom-Up Integration Testing

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

- There is no single strategy adopted for industry practice.
- Modules can be integrated by combining top-down and bottom-up techniques known as *sandwich integration testing*.
- Selection of an integration testing strategy depends on software characteristics and sometimes project schedules.
- In general, sandwich testing strategy that uses top-down tests for upper levels of the program structure with bottom-up tests for subordinate levels is the best compromise.
- The practical approach for adopting sandwich testing is driven by the following factors:
  1. Priority – Integrate the modules based on priority of modules.
  2. Availability – The module which is ready to integrate will be integrated and tested first.

Pros and Cons of Decomposition-Based Integration

- Decomposition-based integration techniques are better for monitoring the progress of integration testing along the decomposition tree.
- If there is failure in this integration, the first suspect goes to the recently added module, as the modules are integrated one by one either in top-down or bottom-up sequence.
- Thus, debugging is easy in decomposition-based integration.
- However, there is more effort required in this type of integration, as stubs and drivers are needed for testing.
- The integration testing effort is computed as the number of test sessions. A test session is one set of test cases for a specific configuration.
- The total number of test sessions in a decomposition-based integration is computed as: **Number of test sessions = nodes – leaves + edges.**

7.2.2 Call Graph-Based Integration

- A call graph is a directed graph, wherein the nodes are either modules or units, and a directed edge from one node to another means one module has called another module.
- The call graph can be captured in a matrix form which is known as the *adjacency matrix*.
- For example, the figure shows how one unit calls another and its adjacency matrix is shown.

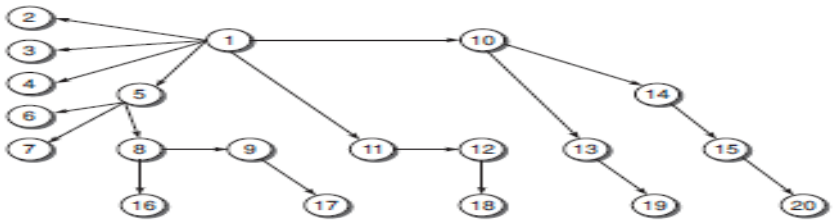


Figure 7.9 Example call graph

	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																				

Figure 7.10 Adjacency matrix

- This integration avoids the efforts made in developing the stubs and drivers.
- There are two types of integration testing based on call graph. A) Pair-wise Integration b) Neighbourhood integration.
- **Pair-wise Integration** - consider only one pair of calling and called modules at a time for integration, and total test sessions which will be equal to the sum of all edges in the call graph. For the given example, **19 sessions** are required
- **Neighbourhood Integration -** consider neighbourhoods of a node in the call graph for integration. The neighbourhood for a node is the immediate predecessor /successor nodes.

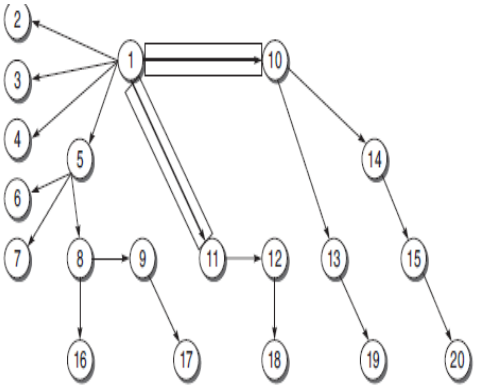


Figure 7.11 Pair-wise integration

Table 7.2 Neighbourhood integration details

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

The total test sessions in neighbourhood integration can be calculated as:

Neighbourhoods = nodes – sink nodes

= 20 – 10

= 10

7.2.3 Path-Based Integration

- It focus on interactions among system units
- It Combine structural and behavioral type of testing for integration testing.
- Source Node: an instruction in the module at which the execution starts or resumes
- Sink Node: an instruction in a module at which the execution terminates
- Module Execution Path (MEP) Message: a path consisting of a set of executable statements within a module like in a flow graph
- MM-Path: a path consisting of MEPs and messages
- MM-Path Graph: an extended flow graph where nodes are MEPs and edges are messages.

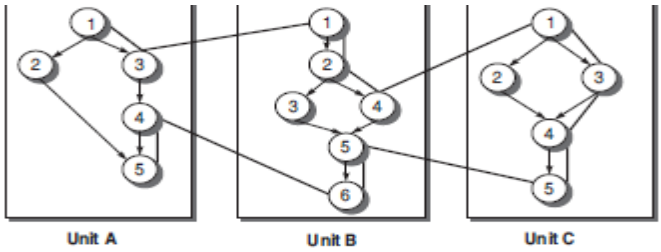


Figure 7.12 MM-path

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>

The MM-path graph for this example is shown in Fig. 7.13.

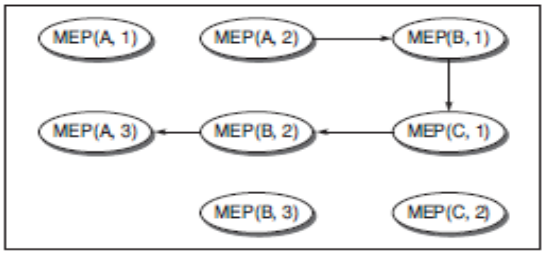


Figure 7.13 MEP graph

7.3 FUNCTION TESTING

- Function testing is the *process of attempting to detect discrepancies between the functional specifications of software and its actual behaviour*.
- Function test is used to measure the quality of the functional (business) components of the system.
- Functional tests verify that the system behaves correctly from the user/business perspective and functions according to the requirements, models, or any other design paradigm used to specify the application.
- The function test must determine if each component or business event:
    - performs in accordance to the specifications,
    - responds correctly to all conditions that may present themselves by incoming events/data,
    - moves data correctly from one business event to the next (including data stores), and
    - is initiated in the order required to meet the business objectives of the system.
- To keep a record of function testing, function coverage metric is used. Function coverage can be measured with a *function coverage matrix*. It keeps track of those functions that exhibited the greatest number of errors.
- The primary processes/deliverables for requirements based function test are:
    - Test planning - defines the scope, schedule, test plan (document) and a test schedule (work plan) and deliverables for the function test cycle.
    - Partitioning/functional decomposition - is the breakdown of a system into its functional components
    - Requirement definition - specified requirements in the form of proper documents
    - 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 - A function coverage matrix is prepared to track which functions are being tested through which test cases.

Table 7.4 Function coverage matrix

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

    - Test case execution - test cases executed and the results are recorded

7.4 SYSTEM TESTING

- *System testing is the process of attempting to demonstrate that a program or system does not meet its original requirements and objectives, as stated in the requirement specification.*
- System testing is a series of tests to test the whole system on various grounds like performance, security, maximum load, etc.
- After passing through these tests, the resulting system is a system which is ready for acceptance testing.
- Design the system test by analysing the objectives; formulate test cases by analysing the user documentation
- There is no identifiable methodology to prepare test cases, so distinct categories of system test cases are taken.

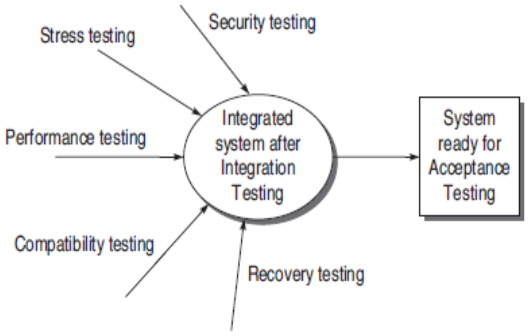


Figure 7.14 System testing

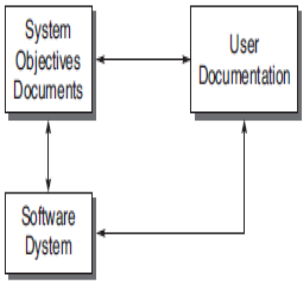


Figure 7.15 Design the system by analysing the objectives

7.4.1 Categories of System Tests

Recovery Testing

- It is the ability of a system to restart operations after the recovery from failure.
- *Recovery testing is the activity of testing how well the software is able to recover from crashes, hardware failures, and other similar problems.*
- Systems (e.g. operating system, database management systems, etc.) must recover to a known state from programming errors, hardware failures, data errors, or any disaster in the system.
- Recovery tests would determine if the system can return to a well-known state, and that no transactions have been compromised.
- A check point system can be used to state the safe position of transactions of system.
- During recovery testing the testers should work on
    - ***Restart*** - If there is a failure, the most recent checkpoint record is to be retrieved and the system is initialized to the states in the checkpoint record and begins to process new transactions.
    - ***Switchover*** - in case of failure of one component, the standby takes over the control. The ability of the system to switch to a new component must be tested.
- A good way to perform recovery testing is under maximum load.

Security Testing

- Security is a protection system that is needed to assure the customers that their data will be protected.
- Security may include controlling access to data, encrypting data in communication, ensuring secrecy of stored data, auditing security events, etc.
- The effects of security breaches could be extensive and can cause loss of information, corruption of information, misinformation, privacy violations, denial of service, etc.
- **Types of Security Requirements** While performing security testing, the following security requirements must be considered -
    - Each functional requirement, most likely, has a specific set of related security issues to be addressed in the software implementation. For example, the log-on must specify the number of retries allowed the action to be taken if the log-on fails, and so on.
    - A software project has security issues that are global in nature, related to the application's architecture and overall implementation. For example, a Web application may have a global requirement that all private customer data of any kind is stored in encrypted form in the database.

- **Security vulnerabilities** Vulnerability is an error that an attacker can exploit.
    - Bugs at the implementation level, such as local implementation errors or inter procedural interface errors
    - Design-level mistakes – hardest category to identify - Examples: problem in error-handling, unprotected data channels, incorrect or missing access control mechanisms, and timing errors especially in multithreaded systems
- **How to perform security testing -** 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.
- **Risk management and security testing** Software security practitioners perform many different tasks to manage software security risks, including:
    - □ Creating security abuse/misuse cases
    - □ Listing normative security requirements
    - □ Performing architectural risk analysis
    - □ Building risk-based security test plans
    - □ Wielding static analysis tools
    - □ Performing security tests
    - Based on design-level risk analysis and ranking of security related risks, security test plans are prepared which guide the security testing.
    - Thus, security testing must necessarily involve two diverse approaches:
        - □ Testing security mechanisms to ensure that their functionality is properly
        - implemented
        - □ Performing risk-based security testing motivated by understanding and
        - simulating the attacker's approach
- **Elements of security testing** The basic security concepts that need to be covered by security testing are discussed below:
    - ***Confidentiality*** A security measure which protects against the disclosure of information to parties other than the intended recipient.
    - ***Integrity*** A measure intended to allow the receiver to determine that the information which it receives has not been altered in transit or by anyone other than the originator of the information.
    - ***Authentication*** A measure designed to establish the validity of a transmission, message, or originator. It allows the receiver to have confidence that the information it receives originates from a specific known source.
    - ***Authorization*** It is the process of determining that a requester is allowed to receive a service or perform an operation. Access control is an example of authorization.
    - ***Availability*** It assures that the information and communication services will be ready for use when expected. Information must be kept available for authorized persons when they need it.
    - ***Non-repudiation*** A measure intended to prevent the later denial that an action happened, or a communication took place, etc. In communication terms, this often involves the interchange of authentication information combined with some form of provable timestamp.

Performance Testing

- Performance testing is to test the run-time performance of the software on the basis of various performance factors. They may be in terms of memory use, response time, throughput, and delays.
- Ex: for a Web application, you need to know at least two things: (a) expected load in terms of concurrent users or HTTP connections and (b) acceptable response time.
- The following tasks must be done for this testing:
    - □ Decide whether to use internal or external resources to perform tests, depending on in-house expertise (or the lack of it).

- Gather performance requirements (specifi cations) from users and/or business analysts.
- Develop a high-level plan (or project charter), including requirements, resources, timelines, and milestones.
- Develop a detailed performance test plan (including detailed scenarios and test cases, workloads, environment info, etc).
- Choose test tool(s).
- Specify test data needed.
- Develop detailed performance test project plan, including all dependencies and associated timelines.
- Configure the test environment (ideally identical hardware to the production platform), router configuration, deployment of server instrumentation, database test sets developed, etc.
- Execute tests, probably repeatedly (iteratively), in order to see whether any unaccounted factor might affect the results.

Load Testing

- *When a system is tested with a load that causes it to allocate its resources in maximum amounts, it is called load testing.*
- Through load testing, we are able to determine the maximum sustainable load the system can handle.
- Load is varied from a minimum (zero) to the maximum level the system can sustain without running out of resources.

Stress Testing

- Stress testing is also a type of load testing, but the difference is that the system is put under loads beyond the limits so that the system breaks.
- 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
- Stress testing is important for real-time systems where unpredictable events may occur, resulting in input loads that exceed the values described in the specification, and the system cannot afford to fail due to maximum load on resources.
- Therefore, in real-time systems, the entire threshold values and system limits must be noted carefully. Then, the system must be stress-tested on each individual value.
- Stress testing demands the amount of time it takes to prepare for the test and the amount of resources consumed during the actual execution of the test.

Usability Testing

- This type of system testing is related to a system's presentation rather than its functionality.
- The goal of usability testing is to verify that intended users of the system are able to interact properly with the system while having a positive and convenient experience.
- *Usability testing identifies discrepancies between the user interfaces of a product and the human engineering requirements of its potential users.*
- What the user wants or expects from the system can be determined using several ways are:

- Area experts
- Group meetings
- Surveys
- Analyse similar products
- Ease of use
- Interface steps
- Response time
- Help System
- Error messages

Compatibility/Conversion/Configuration Testing

- Compatibility testing is to check the compatibility of a system being developed with different operating system, hardware and software configuration available, etc.
- Configuration testing allows developers/testers to evaluate system performance and availability when hardware exchanges and reconfigurations occur.
- Some guidelines for compatibility testing are
- ***Operating systems-*** The specifications must state all the targeted end-user operating systems on which the system being developed will be run.
- ***Software/Hardware -***The product may need to operate with certain versions of Web browsers, with hardware devices such as printers, or with other softwares such as virus scanners or word processors.
- ***Conversion testing-*** Compatibility may also extend to upgrades from previous versions of the software.
- ***Ranking of possible configurations -***Since there will be a large set of possible configurations and compatibility concerns, the testers must rank the possible configurations in order, from the most to the least common, for the target system.
- ***Identification of test cases -*** select the most representative set of test cases that confirms the application's proper functioning on a particular platform.
- ***Updating the compatibility test cases -***The compatibility test cases must also be continually updated

7.5 ACCEPTANCE TESTING

- *Acceptance testing is the formal testing conducted to determine whether a software system satisfies its acceptance criteria and to enable buyers to determine whether to accept the system or not.*
- The purpose of acceptance testing is to give the end user a chance to provide the development team with feedback as to whether or not the software meets their needs.
- User acceptance testing should be carried out by end-users. Acceptance testing is designed to:
    - □ Determine whether the software is fi t for the user.
    - □ Making users confident about the product.
    - □ Determine whether a software system satisfies its acceptance criteria.
    - □ Enable the buyer to determine whether to accept the system or not.
    - **Entry Criteria**
        - □ System testing is complete and defects identified are either fixed or documented.
        - □ Acceptance plan is prepared and resources have been identified.
        - □ Test environment for the acceptance testing is available.
    - **Exit Criteria**
        - □ Acceptance decision is made for the software.
        - □ In case of any warning, the development team is notifi ed.

- **Types of Acceptance Testing**
- Acceptance testing is classified into the following two categories:
  - *Alpha Testing* Tests are conducted at the development site by the end users. The test environment can be controlled a little in this case.
  - *Beta Testing* Tests are conducted at the customer site and the development team does not have any control over the test environment.

7.5.1 Alpha Testing

- Alpha is the test period during which the product is complete and usable in a test environment, but not necessarily bug-free
- Therefore, alpha testing is typically done for two reasons:
- (i) To give confidence that the software is in a suitable state to be seen by the customers (but not necessarily released).
- (ii) To find bugs that may only be found under operational conditions. Any other major defects or performance issues should be discovered in this stage.
- Since alpha testing is performed at the development site, testers and users together perform this testing. Therefore, the testing is in a controlled manner such that if any problem comes up, it can be managed by the testing team.
- **Entry Criteria to Alpha**
  - All features are complete/testable (no urgent bugs).
  - High bugs on primary platforms are fixed/verified.
  - 50% of medium bugs on primary platforms are fixed/verified.
  - All features have been tested on primary platforms.
  - Performance has been measured/compared with previous releases (user functions).
  - Alpha sites are ready for installation.
- **Exit Criteria from Alpha**
  - After alpha testing, we must:
  - Get responses/feedbacks from customers.
  - Prepare a report of any serious bugs being noticed.
  - Notify bug-fixing issues to developers.

7.5.2 Beta Testing

- Versions of the software, known as beta-versions, are released to a limited audience outside the company.
- The software is released to groups of people so that further testing can ensure the product has few or no bugs.
- Sometimes, beta-versions are made available to the open public to increase the feedback field to a maximal number of future users.
- **Entry Criteria to Beta**
  - Positive responses from alpha sites.
  - Customer bugs in alpha testing have been addressed.
  - There are no fatal errors which can affect the functionality of the software.
  - Secondary platform compatibility testing is complete.
  - Regression testing corresponding to bug fixes has been done.
  - Beta sites are ready for installation.
- **Guidelines for Beta Testing**
  - Don’t expect to release new builds to beta testers more than once every two weeks.
  - Don’t plan a beta with fewer than four releases.
  - If you add a feature, even a small one, during the beta process, the clock goes back to the beginning of eight weeks and you need another 3–4 releases.
- **Exit Criteria from Beta**

  After beta testing, we must:
  - Get responses/feedbacks from the beta testers.
  - Prepare a report of all serious bugs.
  - Notify bug-fixing issues to developers

Regression Testing

8.1 Progressive vs. Regressive Testing

- A system under test (SUT) is said to *regress* if
  - a modified component fails, or
  - a new component, when used with unchanged components, causes failures in the unchanged components by generating side-effects or feature interactions.
- Regression testing is not another testing activity. Rather, it is the re-execution of some or all of the already developed test cases.
- *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 testing can be defined as the *software maintenance task performed on a modified program to instill confidence that changes are correct and have not adversely affected the unchanged portions of the program*

8.2 Regression Testing produces Quality Software

- Indeed, the importance of regression testing is well-understood for the following reasons:
  - It validates the parts of the software where changes occur.
  - It validates the parts of the software which may be affected by some changes, but otherwise unrelated.
  - It ensures proper functioning of the software, as it was before changes occurred.
  - It enhances the quality of software, as it reduces the risk of high-risk bugs.

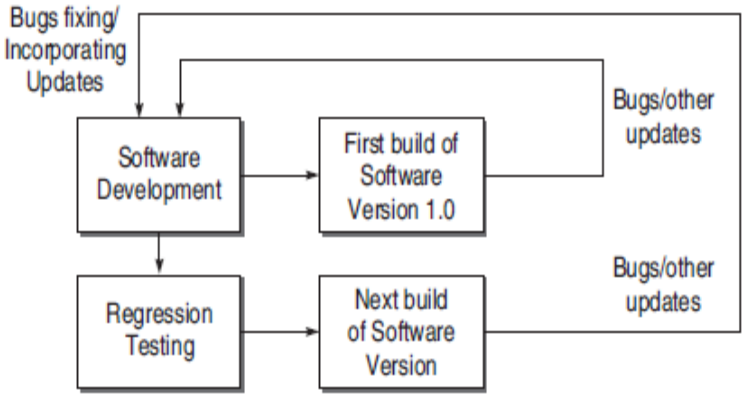


Figure 8.1 Regression testing produces Quality Software

8.4 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.
- ***If finds other related bugs*** - It may be possible that the developer has fixed only the symptoms of the reported bugs without fixing the underlying bug. Moreover, there may be various ways to produce that bug. Therefore, 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, regression tests are necessary to check the influence of changes in one part on other parts of the program.

8.5 WHEN IS REGRESSION TESTING DONE?

1. Software Maintenance

Corrective maintenance Changes made to correct a system after a failure has been observed (usually after general release).

Adaptive maintenance Changes made to achieve continuing compatibility with the target environment or other systems.

Perfective maintenance Changes designed to improve or add capabilities.

Preventive maintenance Changes made to increase robustness, maintainability, portability, and other features.

2. 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.

3. First Step of Integration

Re-running accumulated test suites, as new components are added to successive test configurations, builds the regression suite incrementally and reveals regression bugs.

4. Compatibility Assessment and Benchmarking

Some test suites are 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.

8.6 REGRESSION TESTING TYPES

- ***Bug-Fix regression*** **-** This testing is performed after a bug has been reported and fixed. Its goal is to repeat the test cases that expose the problem in the first place.
- ***Side-Effect regression/Stability regression*** It involves retesting a substantial part of the product. The goal is to prove that the change has no detrimental effect on something that was earlier in order. It tests the overall integrity of the program, not the success of software fixes.

8.7 DEFINING REGRESSION TEST PROBLEM

Let us first define the notations used in regression testing before defining the regression test problem.

- *P* denotes a program or procedure,
- *P'* denotes a modified version of *P*,
- *S* denotes the specification for program *P*,
- *S'* denotes the specification for program *P'*,
- *P(i)* refer to the output of *P* on input *i*,
- *P'*(*i*) refer to the output of *P'* on input *i*, and
- $T = \{t_1, \ldots t_n\}$ denotes a test suite or test set for *P*.
- Given a program *P*, its modified version *P'*, and a test set *T* that was used earlier to test *P*; find a way to utilize *T* to gain sufficient confidence in the correctness of *P'*.

8.8 REGRESSION TESTING TECHNIQUES

- ***Regression test selection technique*** **-**This technique attempt to reduce the time required to retest a modified program by selecting some subset of the existing test suite.
- ***Test case 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 with lower priority. There are two types of prioritization:
    - **(a)General Test Case Prioritization** For a given program *P* and test suite *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 knowledge of the changes made in *P*.

- ***Test suite reduction technique*** -It reduces testing costs by permanently eliminating redundant test cases from test suites in terms of codes or functionalities exercised.

8.8.1 SELECTIVE RETEST TECHNIQUE

- Selective retest techniques attempt to reduce the cost of testing by identifying the portions of *P'* that must be exercised by the regression test suite.
- The objective of selective retest technique is *cost reduction*.
- It is the process of selecting a subset of the regression test suite that tests the changes.
- 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 uses the information about changes to select test cases.

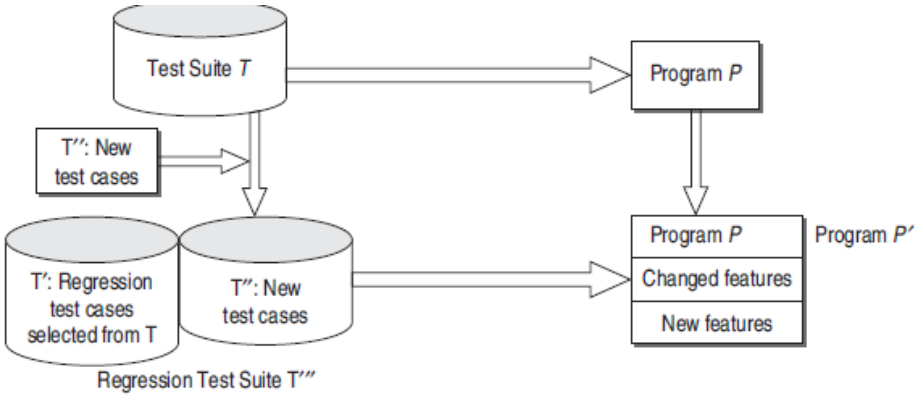


Figure 8.2 Selective Retest Technique

1. Select *T'* subset of *T*, a set of test cases to execute on *P'*.
2. Test *P'* with *T'*, establishes correctness of *P'* with respect to *T'*.
3. If necessary, create *T* , a set of new functional or structural test cases for *P'*.
4. Test *P'* with *T' '*, establishes 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 ' '*.

Each of these steps involves the following important problems:

Regression test selection problem Step 1 involves this problem. The problem is to select a subset *T'* of *T* with which *P'* will be tested.

Coverage identification problem Step 3 addresses this problem. The problem is to identify portions of *P '* or its specifications that requires additional testing.

Test suite execution problem Steps 2 and 4 address the test suite execution problem. The problem is to execute test suites efficiently and checking test results for correctness.

Test suite maintenance problem Step 5 addresses this problem. The problem is to update and store test information.

Regression Test Selection Techniques

A variety of regression test selection techniques have been described in the research literature. We consider the following selective retest 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*.

- ***Dataflow techniques*** Dataflow coverage-based regression test selection techniques select test cases that exercise data interactions that have been affected by modifications.
- ***Safe techniques*** - 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*** - randomly select a predetermined number of test cases from *T or* select test cases based on 'intuitions'.
- ***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 Techniques

The following categories for evaluating the regression test selection techniques:

Inclusiveness It measures the extent to which M chooses modification-revealing test from T for inclusion in T', where M is a regression test selection technique.

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 and P' and T is:

1. $\text{INCL}(M) = (100 \times (m/n)\%$, if $n \neq 0$
2. $\text{INCL}(M)\% = 100\%$, if $n = 0$

For example, if T contains 100 tests of which 20 are modification-revealing for P and P', and M selects 4 of these 20 tests, then M is 20% 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.

If for all P, P', and T, M is 100% inclusive relative to P, P', and T, then M is safe.

Precision It measures the extent to which M omits tests that are non-modification-revealing. 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 given by

$\text{Precision} = 100 \times (m/n)$ % if $n \neq 0$

$\text{Precision} = 100$ % if $n = 0$

For example, if T contains 50 tests of which 22 are non-modification-revealing for P and P', and M omits 11 of these 22 tests, then M is 50% 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 The efficiency of regression test selection techniques is measured 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 depends on the size of the test suite that a technique selects, and on the computational cost of that technique.

8.8.2 Regression Test Prioritization

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. By prioritizing the execution of a regression test suite, these methods reveal important defects in a software system earlier in the regression testing process.

This approach can be used along with the previous selective retest technique. The steps for this approach are given below:

1. Select T' from T, a set of test 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' with respect to 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' with respect to T''.
6. Create T''', a new test suite for P', from T, T'_p, and T''.