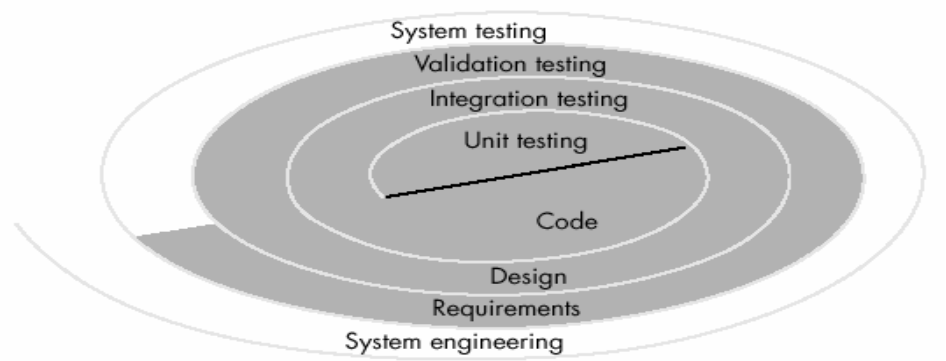
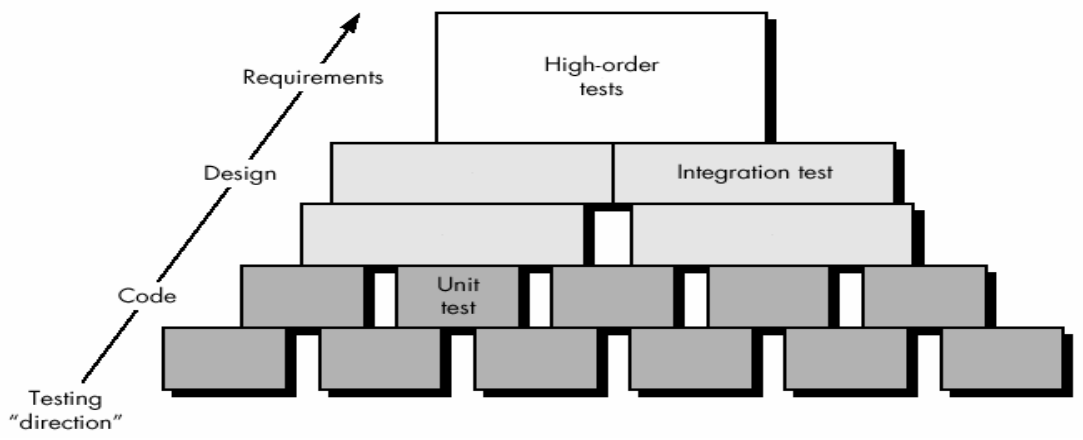
# *Software Testing Strategies*




Testing of software is actually a series of 4 steps.
**1.** The first step individually tests each unit of the software, ensuring that it functions properly as a unit. Hence, the name *unit testin*g.
**2.** Next, components must be assembled or integrated to form the complete software package. So, integration testing addresses these issues.
**3.** After the software has been integrated (constructed), a set of high-order tests are conducted. Validation testing provides final assurance that software meets all functional, behavioral, and performance requirements.
**4.** The last high-order testing step falls outside the boundary of software engineering and into the broader context of computer system engineering. Software, once validated, must be combined with other system elements (e.g., hardware, people, and databases). System testing verifies that all elements mesh properly and that overall system function/performance is achieved.




## Verification and Validation

Software testing is one element of a broader topic that is often referred to as verification *and* validation (V&V).
**Verification: "Are we building the product right?"**
**Validation: "Are we building the right product?"**

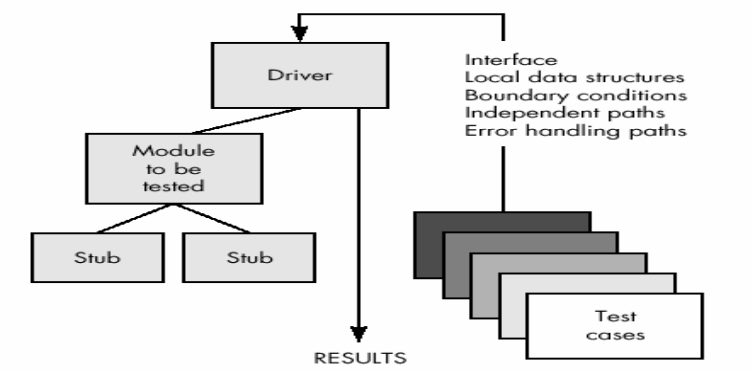
**Verification** refers to the set of activities that ensure that software correctly implements a specific function.
**Validation** refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements.
The definition of verification and validation encompasses many activities that we refer to as software quality assurance.
Software engineering activities provide a foundation from which quality is built.
Analysis, design, coding methods act to enhance the quality by providing uniform techniques and predictable results.
Formal technical reviews help to ensure the quality of the products. Throughout the process, measures and controls are applies to every element of software configuration. These help to maintain uniformity.
**Testing is the last phase in which quality is accessed and errors are uncovered.**

## UNIT TESTING

- Unit testing focuses verification effort on the smallest unit of the software
- Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module.
- The relative complexity of tests and uncovered errors is limited by the limited scope established for unit testing.
- A unit test is white-box oriented
- Unit tests are conducted in parallel for multiple components.

**TESTS CONDUCTED**
1. The *module interface is tested* to ensure that information flows properly into and out of the unit under test.
2. The *local data structure is tested* to ensure that data stored temporarily maintains its integrity during all the steps of execution.
3. *Boundary conditions are tested* to establish that the module operates properly at the established boundaries.
4. *All independent paths through the module are exercised* to ensure that all the statements are executed at least once.
5. Finally *error handling paths are also tested*.

- Unit testing is normally considered as an addition to coding. After the source level code is developed, reviewed and verified for syntax, unit test case design begins.



**UNIT TEST PROCEDURES**

- A component is not a stand-alone program
- Hence a driver and/or stub software must be developed for each unit test.
- **Driver –** It is nothing more than a "main program" that accepts test case data, passes such data to the component (to be tested), and prints relevant results.
- **Stubs -** They serve to replace modules that are subordinate (called by) the component to be tested. A stub or "dummy subprogram" uses the subordinate module's interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.
- Drivers and stubs represent overhead. That is, both are software that must be written but that are not delivered with the final software product.

## INTEGRATION TESTING

*"If they all work individually, why do you doubt that they'll work when we put them together?"*
The problem, of course, is "putting them together"—interfacing.

- Data can be lost across an interface
- One module can have an adverse affect on another
- Sub functions, when combined, may not produce the desired major function
- Individually acceptable errors may be magnified to unacceptable levels
- Global data structures can present problems.

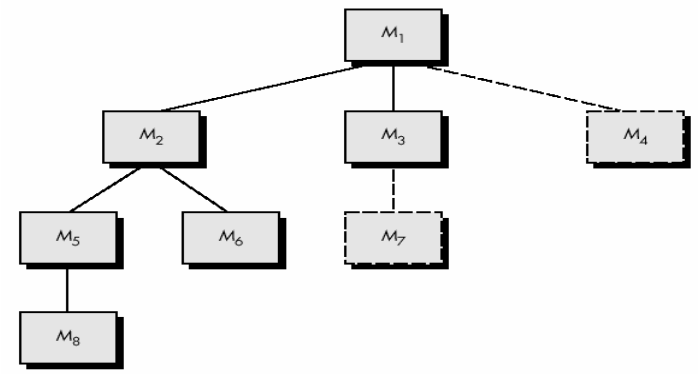
- Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing
- The objective is to take unit tested components and build the entire program.
- **Non-incremental integration** that is, to construct the program using a "big bang" approach. All components are combined in advance. The entire program is tested as a whole. A set of errors is encountered. Correction is difficult because isolation of causes is complicated as the program size is huge.
- **Incremental integration** is opposite to the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied.

## 1. Top-down Integration

- Top-down integration testing is an incremental approach to construction of program structure.
- Modules are integrated by moving downward through the control hierarchy, beginning with the main control module (main program).
- Modules subordinate (and ultimately subordinate) to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

☞ **Depth-first integration** would integrate all components on a major control path of the structure. Selection of a path is arbitrary and depends on application-specific characteristics. For example, selecting the left-hand path, M 1, M 2, M 5, M 8.

- **Breadth-first integration** incorporates all components directly subordinate at each level, moving across the structure horizontally. Components M 2, M 3, and M 4 would be integrated first. The next control level, M 5, M 6.



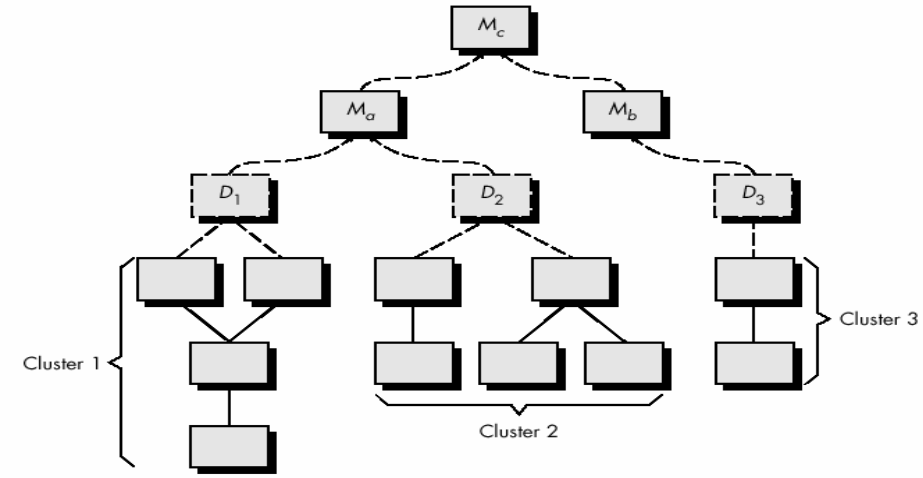

The integration process is performed in a series of five steps:
1. The main control module is used as a test driver and stubs are substituted for all components directly subordinate to the main control module.
2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.
3. Tests are conducted as each component is integrated.
4. On completion of each set of tests, another stub is replaced with the real component.
5. Regression testing may be conducted to ensure that new errors have not been introduced.

- The top-down integration strategy verifies major control or decision points early in the test process. In a well-factored program structure, decision making occurs at upper levels in the hierarchy and is therefore encountered first.
- One of the common problems with top – down strategy is when, processing in the lower levels of the hierarchy is required to adequately test the upper levels (solution is bottom up).

## 2. Bottom-up Integration

- Bottom-up integration testing, as its name implies, begins construction and testing with *atomic modules* (i.e., components at the lowest levels in the program structure).
- Because components are integrated from the bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated.

A bottom-up integration strategy may be implemented with the following steps:
1. Low-level components are combined into clusters (sometimes called *build*s) that perform a specific software sub function.
2. A driver (a control program for testing) is written to coordinate test case input and output.
3. The cluster is tested.
4. Drivers are removed and clusters are combined moving upward in the program structure.




Integration follows the pattern illustrated in Figure. Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to Ma. Drivers D 1 and D 2 are removed and the clusters are interfaced directly to Ma.

## REGRESSION TESTING

- Each time a new module is added as part of integration testing, the software changes.
- New data flow paths are established, new I/O may occur, and new control logic is invoked.
- These changes may cause problems with functions that previously worked flawlessly.
- In the context of an integration test strategy, ***regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects.***
- Regression testing is the activity that helps to ensure that changes do not introduce unintended behavior or additional errors.
- Regression testing may be conducted manually, by re-executing a subset of all test cases or using automated *capture/playback tools.*
- **Capture/playback tools** enable the software engineer to capture test cases and results for subsequent playback and comparison.

## SMOKE TESTING

- *Smoke testing* is an integration testing approach that is commonly used when "shrink-wrapped" software products are being developed. It is designed as a pacing mechanism for time-critical projects, allowing the software team to assess its project on a frequent basis.

In essence, the smoke testing approach encompasses the following activities:
1. Software components that have been translated into code are integrated into a "build." A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
2. A series of tests is designed to expose errors that will keep the build from properly performing its function. The intent should be to uncover "show stop-per" errors that have the highest likelihood of throwing the software project behind schedule.

3. The build is integrated with other builds and the entire product (in its current form) is smoke tested daily. The integration approach may be top down or bottom up.

Smoke testing provides a number of benefits when it is applied on complex, time-critical software engineering projects:

- ***Integration risk is minimized***. Because smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early, thereby reducing the likelihood of serious schedule impact when errors are uncovered.
- ***The quality of the end-product is improved***. Because the approach is construction (integration) oriented, smoke testing is likely to uncover both functional errors and architectural and component-level design defects. If these defects are corrected early, better product quality will result.
- ***Error diagnosis and correction are simplified.*** Like all integration testing approaches, errors uncovered during smoke testing are likely to be associated with "new software increments"—that is, the software that has just been added to the build(s) is a probable cause of a newly discovered error.
- ***Progress is easier to assess.*** With each passing day, more of the software has been integrated and more has been demonstrated to work. This improves team morale and gives managers a good indication that progress is being made.

## VALIDATION TESTING

Software validation is achieved through a series of black box test that demonstrate conformity with the requirements. A test plan defines test cases which would establish conformity.
Result of a validation test is either of the two
1. The function or the performance conform to the specifications and are accepted.
2. A deviation from the specifications is observed hence a deficiency list is prepared.

**Review**

An important element of the validation process is a *configuration review.* The intent of the review is to ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail to support software life cycle. The configuration review, sometimes called an ***audit.***

**ALPHA TESTING**

- Conducted at the developer's site by the customer.
- The developers are present during testing.
- The developer records the errors and usage problems.
- Alpha tests are conducted in a controlled environment.

**BETA TESTING**

- Conducted at the customer sites by the end users of the software.
- The developers are not present during testing.
- The customer records all the errors and reports them to the developers at regular intervals.
- Software is used in an uncontrolled environment.

## SYSTEM TESTING

- Software is only one element of a larger computer-based system.
- Ultimately, software is incorporated with other system elements (e.g., hardware, people, information), and a series of system integration and validation tests are conducted.
- These tests fall outside the scope of the software process and are not conducted solely by software engineers. However, steps taken during software design and testing can greatly improve the probability of successful software integration in the larger system.
- A classic system testing problem is "finger-pointing." This occurs when an error is uncovered, and each system element developer blames the other for the problem.
- System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system.

### 1. Recovery Testing

- Many computer based systems must recover from faults and resume processing within a pre-specified time.
- In some cases, a system must be fault tolerant. In other cases, a system failure must be corrected within a specified period of time or severe economic dam-age will occur.
- *Recovery testing* is a system test that forces the software to fail in a variety of ways and verifies that recovery is properly performed.
- If recovery is automatic (performed y the system itself), re-initialization, check pointing mechanisms, data recovery, and restart are evaluated for correctness.
- If recovery requires human intervention, the mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.

### 2. Security Testing

- Any computer-based system that manages sensitive information and can harm individuals is a target for improper or illegal penetration.
- *Security testing* attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration.
- During security testing, the tester plays the role of the hacker.
- Given enough time and resources, good security testing will ultimately penetrate a system. The role of the system designer is to make penetration cost more than the value of the information that will be obtained.

### 3. Stress Testing

- Stress tests are designed to confront programs with abnormal situations.
- *Stress testing* executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.
- A variation of stress testing is a technique called **sensitivity testing.**
- In some situations a very small range of data contained within the bounds of valid data for a program may cause extreme and even incorrect processing or profound performance degradation.
- Sensitivity testing attempts to uncover data combinations within valid input classes that may cause instability or improper processing.

### 4. Performance Testing

- Software that performs all the required functions but does not conform to the performance requirements is unacceptable.

- Performance testing is designed to test the run time performance of the system which is integrated with the entire system
- Performance testing occurs through the entire testing process.
- But its true measure can be evaluated only when it is integrated with the entire system.

## TEST DATA

The data which is used for testing is called test data.

**1. Live Test Data**

- This data is extracted from the organization files.
- After the system is partially constructed the analysts may ask the users to input the data of their normal activities.
- It is difficult to obtain test data in sufficient amounts for extensive testing.
- Although this is realistic data, it does not help to test the unusual occurrences.

**2. Artificial Test Data**

- Since it is difficult to obtain large volumes of live data, artificial data is created, solely for the purpose of testing.
- Artificial test data and testing tools are maintained in the testing libraries.
- Since they are generated artificially any combination, formats and values of data can be used.

# Short Note on Debugging

Debugging is the art of diagnosing the precise nature of a known error and then correcting the error.

Whenever the test finds an error, debugging is the process that results in the removal of the error. The debugging process attempts to match the symptoms with the cause, thereby leading to the error correction. Debugging may or not always be successful in finding the cause and eliminating it.

It is impossible to present a set of instructions for program debugging. The debugger must generate hypothesis about the observable behavior of the program and test the hypothesis with the hope of finding the error in the system.

The debugging process will always have two outcomes –

1. The cause will be found and corrected.
2. The cause will not be found.

In the second case the person performing debugging may suspect a cause and design test cases to validate his suspicion.

**Characteristics of the Debugging Process**

- Symptom and the cause may be geographically remote ie symptom may appear in one part of the program while cause might be at a site that is far remote.
- Symptom may disappear when another error is corrected.
- Symptom may be caused by non errors( eg rounding off)
- Symptom may cause due to the human errors which are not easily traceable.
- It may be difficult to accurately reproduce the input conditions.
- Symptom may be intermittent eg embedded system, which couple hardware and software inextricably.
- Symptom may be due to the causes distributed across a no of tasks running on different processors.

## APPROACHES AND STRATEGIES

**1) Brute Force**

It is the most common and the least efficient method for isolating the cause of the software error. The methods are used as the last resource. Here memory dumps are taken, run time traces are invoked and program is loaded with the Write statements, in the hope of finding a due to error. It is a tedious and a time consuming task, with no guarantee of results.

**2) Backtracking**

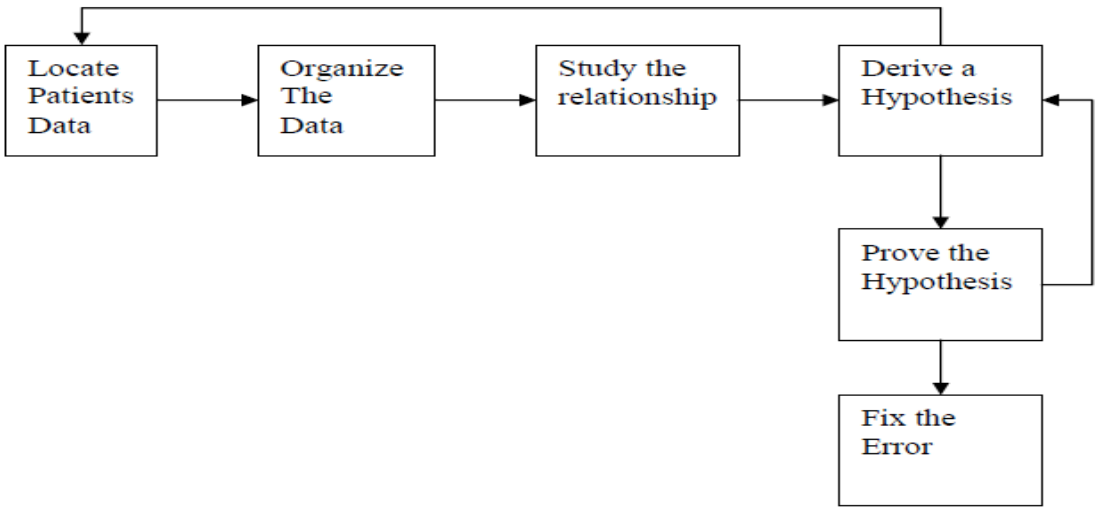
It is a commonly used debugging approach, used generally for small programs.
Beginning at a site where the symptom first uncovered, code is traced backward until the site of cause is found. It cannot be successfully used for complex large programs having multiple data flows.

**3) Inductive Approach**

It comes from the formulation of a single working hypothesis based on the data, on analysis of existing data and on especially collected data to prove or disapprove the working hypothesis.

**Steps Involved**

- **Locate the pertinent data** – It is the enumeration of all that is known about what the program did correctly it did incorrectly. Valuable clues are provided by similar test cases that do not cause the symptom to appear.
- **Organize the data** – It is structuring the pertinent data to allow one to observe patterns. Here it is important to search for contradictions.
- **Device Hypothesis -** This step studies the relationships among the clues and to device using the patterns that might be visible in the structure of the clues, one or more hypothesis about the cause of the error. If a theory cannot be devised, more data is necessary to be obtained. If multiple theories are possible the most probable one is selected first.
- **Prove the Hypothesis -** It is vital to prove the reasonableness of the hypothesis before proceeding. Failure to do so results in fixing of only the symptom of the problem or only a portion of it. Hypothesis is proved by comparing it to the original dues, data and making sure that the hypothesis completely explains the existence of the clues.




**Flow Chart for Inductive Debugging Process**

4) **Deductive Approach**: -

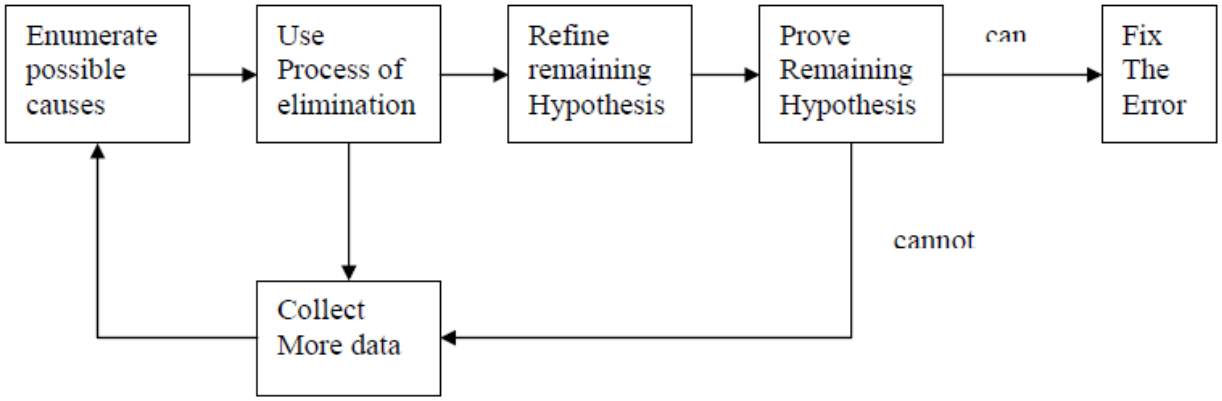
This approach involves listing all causes or hypothesis which seems possible.

Then one by one particular causes are ruled out till a single one remains for validation.

Enumerate all possible causes: It involves making a list of al conceivable causes of the error.

They need to be complete explanation, but theories, which one can use to structure and analyze available data.

1. **Use data to eliminate causes**: careful data analysis is used to eliminate all but one cause. If all are eliminated, additional data is required to devise new theories, If more than one remains, the most probable one is selected first.
2. **Refine the remaining hypothesis**:- Next step is to use available dues to refine the theory i.e., To get down from the general to the specifics of the details. E.g. ("error in handling last transaction in file") is refined to ("last transaction in buffer overlaid with EOF indicator")
3. **Prove the Hypothesis**: - step same as above for inductive approach.




**Flow of Inductive Debugging**