UNIT-V:

Testing Strategies: A strategic approach to software testing, strategies issues, test strategies for O-O software, Validation testing, System testing, the art of Debugging.

Testing Tactics: Software Testing Fundamentals, Black-Box and White-Box Testing, basis path Testing, Control Structure Testing, O-O Testing methods, Testing Methods applicable on the class level, inter class test case design, Testing for Specialized environments, architectures and applications, Testing Patterns.

Product metrics: Software Quality, A Frame work for Product Metrics, Metrics for Analysis Model, Metrics for Design Model, Metrics for source code, Metrics for testing, Metrics for maintenance.

Software Testing Strategies

Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.

Software Testing is One of the important phases of software development. Testing is the process of execution of a program with the intention of finding errors Involves 40% of total project cost.

Who Tests the Software?

- Development engineers
  - Understand the system, but test "gently"
  - Driven by "delivery"
  - Only perform unit tests and integration tests
- Test engineers
  - Need to learn the system, but attempt to break it
  - Driven by "quality"
  - Define test cases, write test specifications, run tests, analyze results
- Customers
  - Driven by "requirements"
  - Determine if the system satisfies the acceptance criteria

Software testing is one element of a broader topic: verification and validation (V&V)

Verification – are we building the product correctly?*

The set of activities to ensure that software correctly implements specific functions

Validation – are we building the correct product?*

The set of activities to ensure that the developed software is traceable to customer requirements

A strategic Approach for Software testing

Testing Strategy is

- A road map that incorporates test planning, test case design, test execution and resultant data collection and execution
- **Validation** refers to a different set of activities that ensures that the software is traceable to the customer requirements.
- V&V encompasses a wide array of Software Quality Assurance

The definition of V&V encompasses many activities that is referred to as software quality assurance (SQA) including

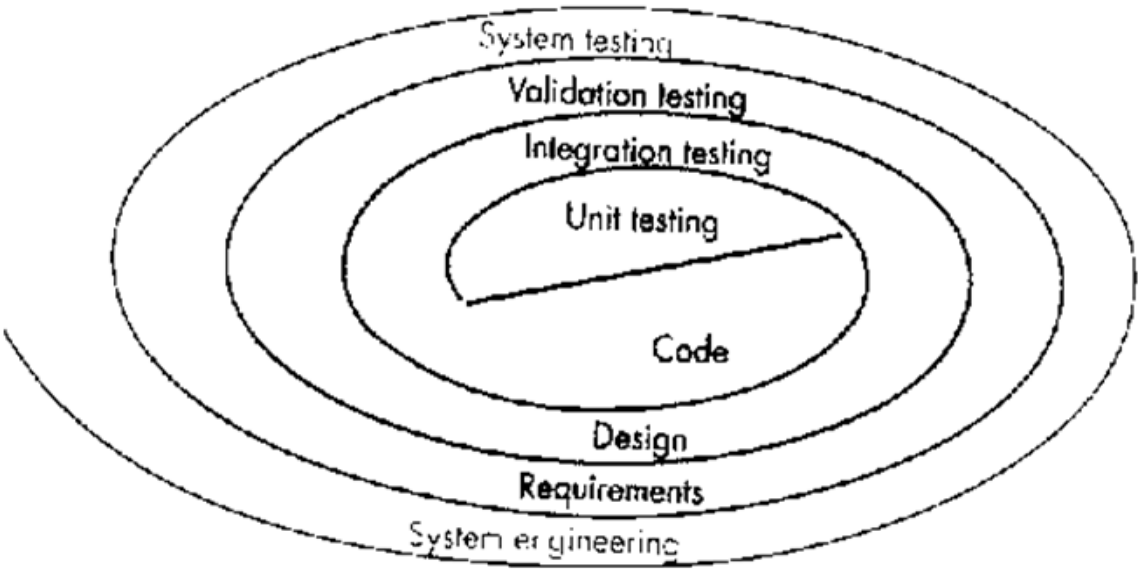
  - Formal technical review
  - Quality and configuration audits
  - Performance monitoring
  - Simulation
  - Feasibility study
  - Documentation review
  - Database review
  - Algorithm analysis
  - Development testing
  - Qualification testing
  - Installation testing

- Perform Formal Technical reviews(FTR) to uncover errors during software development
- Begin testing at component level and move outward to integration of entire component based system.
- Adopt testing techniques relevant to stages of testing
- Testing can be done by software developer and independent testing group
- Testing and debugging are different activities. Debugging follows testing
- Low level tests verifies small code segments.
- High level tests validate major system functions against customer requirements

Testing Strategies for Conventional Software

1)Unit Testing
2)Integration Testing
3)Validation Testing and
4)System Testing

Spiral Representation for Conventional Software

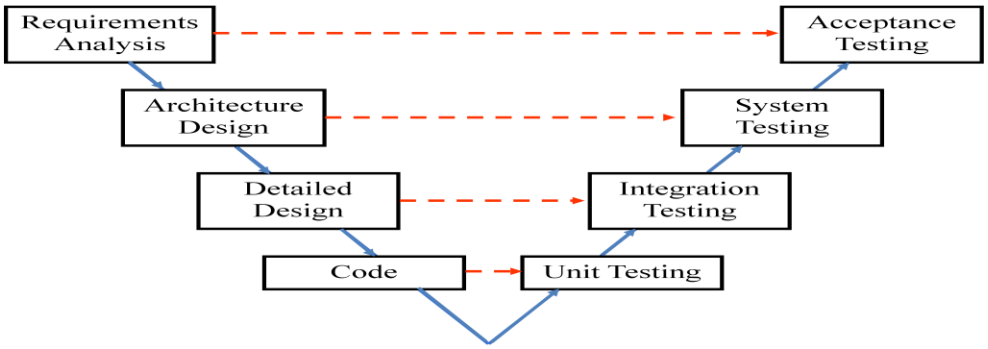


Strategic Issues

- Specify product requirements in a quantifiable manner long before testing.
- State testing objectives explicitly.
- Understand the users of the software and develop a profile for each user category.
- Develop a testing plan that emphasizes “rapid cycle testing.”
- Build “robust” software that is designed to test itself
- Use effective formal technical reviews as a filter prior to testing
- Conduct formal technical reviews to assess the test strategy and test cases themselves.
- Develop a continuous improvement approach for the testing process.
-

The V model

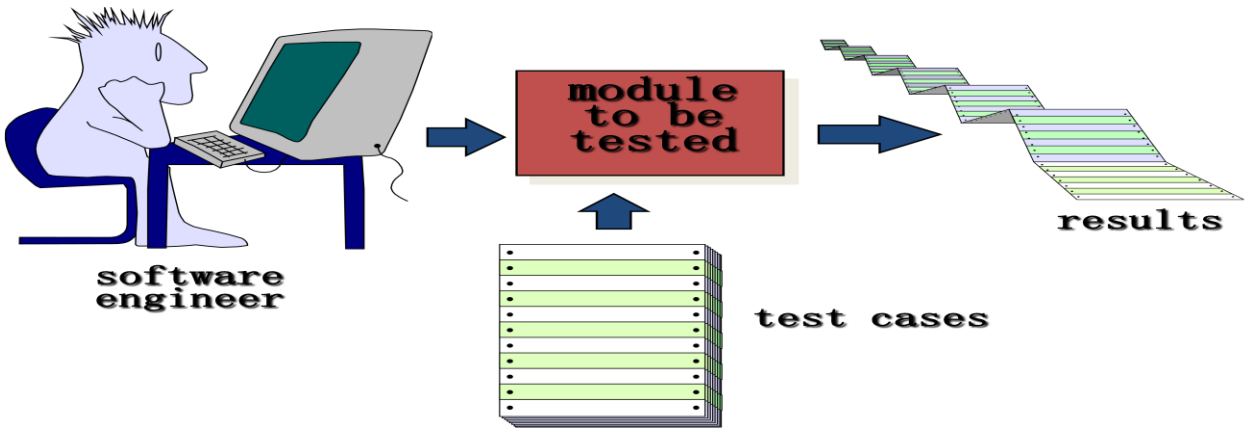
- Emerged in reaction to some waterfall models that showed testing as a single phase following the traditional development phases of requirements analysis, high-level design, detailed design and coding.
- The V model portrays several distinct testing levels and illustrates how each level addresses a different stage of the software lifecycle.
- The V shows the typical sequence of development activities on the left-hand (downhill) side and the corresponding sequence of test execution activities on the right-hand (uphill) side.



- The V model is valuable because it highlights the existence of several levels of testing and delineates how each relates to a different development phase:
    - Unit testing: concentrates on each unit (i.e., component) of the software (white box)
    - Integration testing: focuses on design and the construction of the software architecture (black box, limited amount of white box)
    - System testing: verifies that all elements mesh properly and that overall system function/performance is achieved.
    - Acceptance testing: are ordinarily performed by the business/users to confirm that the product meets the business requirements.

(1). Unit Testing

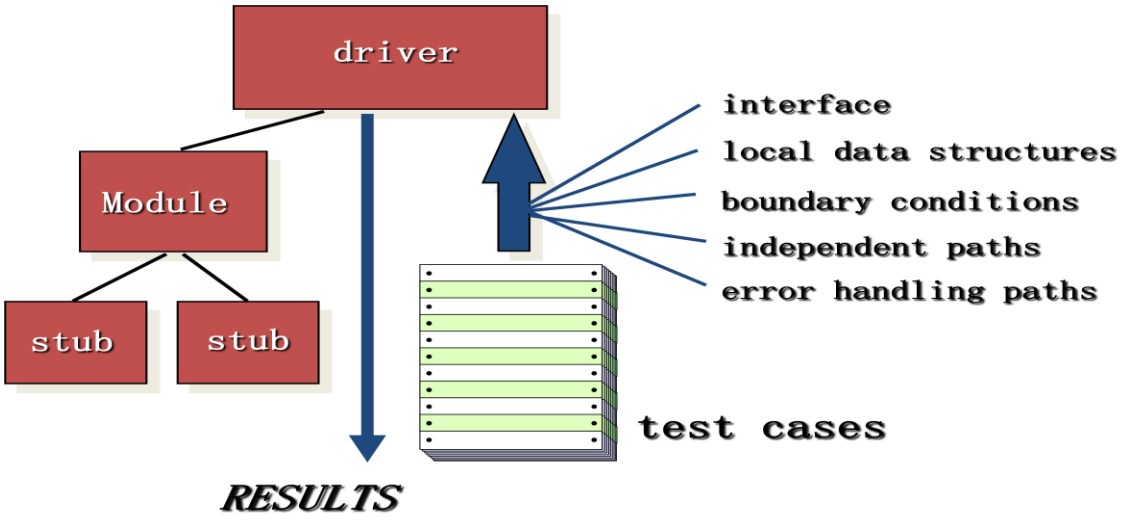
Unit testing focuses verification effort on the smallest unit of software design—the software component or module. Using the component-level design description as a guide, important control paths are tested to uncover errors within the boundary of the module. The relative complexity of tests and uncovered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



The module interface is tested to ensure that information properly flows into and out of the program unit under test. The local data structure is examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm's execution. Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. All independent paths (basis paths) through the control structure are exercised to ensure that all statements in a module have been executed at least once. And finally, all error handling paths are tested.

Unit Test Environment

Because a component is not a stand-alone program, driver and/or stub software must be developed for each unit test. The unit test environment is illustrated in the above figure. In most applications a **driver** is nothing but a **"main program"** that accepts test case data, passes such data to the component (to be tested), and prints relevant results. **Stubs** 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 (formal design is not commonly applied) but that is not delivered with the final software product. If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be adequately unit tested with "simple" overhead software. In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used).

Among the more common errors in computation are:
(1) misunderstood or incor-rect arithmetic precedence, (2) mixed mode operations, (3) incorrect initialization, (4) precision inaccuracy, (5) incorrect symbolic representation of an expression. Com-parison and control flow are closely coupled to one another (i.e., change of flow fre-quently occurs after a comparison).

Test cases should uncover errors such as:
(1) comparison of different data types, (2) incorrect logical operators or precedence, (3) expectation of equality when precision error makes equality unlikely, (4) incorrect comparison of variables, (5) improper or nonexistent loop termination, (6) failure to exit when divergent iteration is encountered, and (7) improperly modified loop variables.

(2). Integration Testing Strategies

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 a program structure that has been dictated by design.

Options:

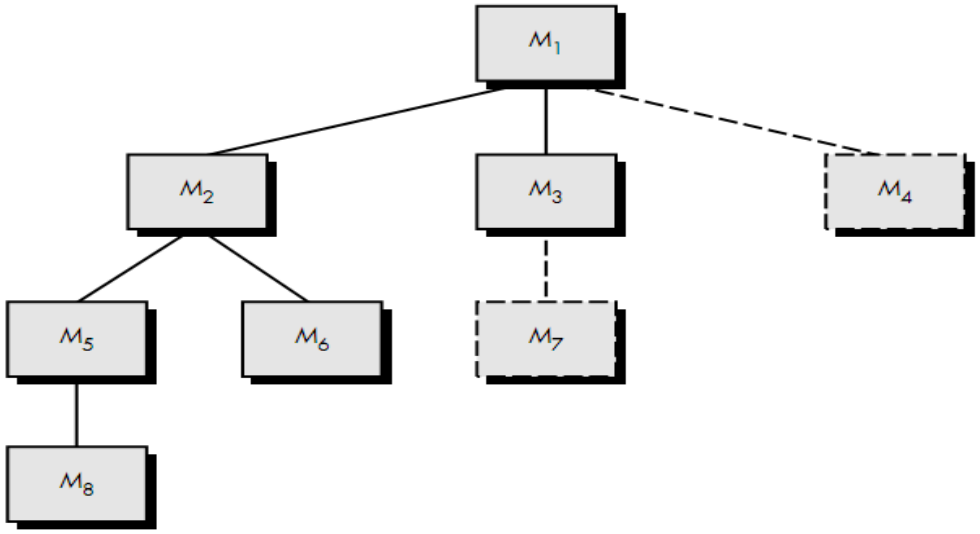
- **The "big bang" approach**
  All components are combined in advance. The entire program is tested as a whole. And chaos usually results! A set of errors is encountered. Correction is difficult because isolation of causes is com-plicated by the vast expanse of the entire program. Once these errors are corrected, new ones appear and the process continues in a seemingly endless loop.

- **An incremental construction strategy**
  - **Top-down integration**
  - **Bottom-up integration**
  - **Sandwich integration**

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 major path is somewhat arbitrary and depends on application-specific characteristics. For example, selecting the left-hand path, components M1, M2 , M5 would be integrated first. Next, M8 or (if neces-sary for proper functioning of M2) M6 would be integrated. Then, the central and right-hand control paths are built. Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally. From the figure, components M2, M3, and M4 (a replacement for stub S4) would be integrated first. The next control level, M5, M6, and so on, follows.

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 process continues from step 2 until the entire program structure is built.

Top-down strategy sounds relatively uncomplicated, but in practice, logistical problems can arise. The most common of these problems occurs when processing at low levels in the hierarchy is required to adequately test upper levels. Stubs replace low-level modules at the beginning of top-down testing.

- Advantages
  - This approach verifies major control or decision points early in the test process
- Disadvantages
  - Stubs need to be created to substitute for modules that have not been built or tested yet; this code is later discarded
  - Because stubs are used to replace lower level modules, no significant data flow can occur until much later in the integration/testing process

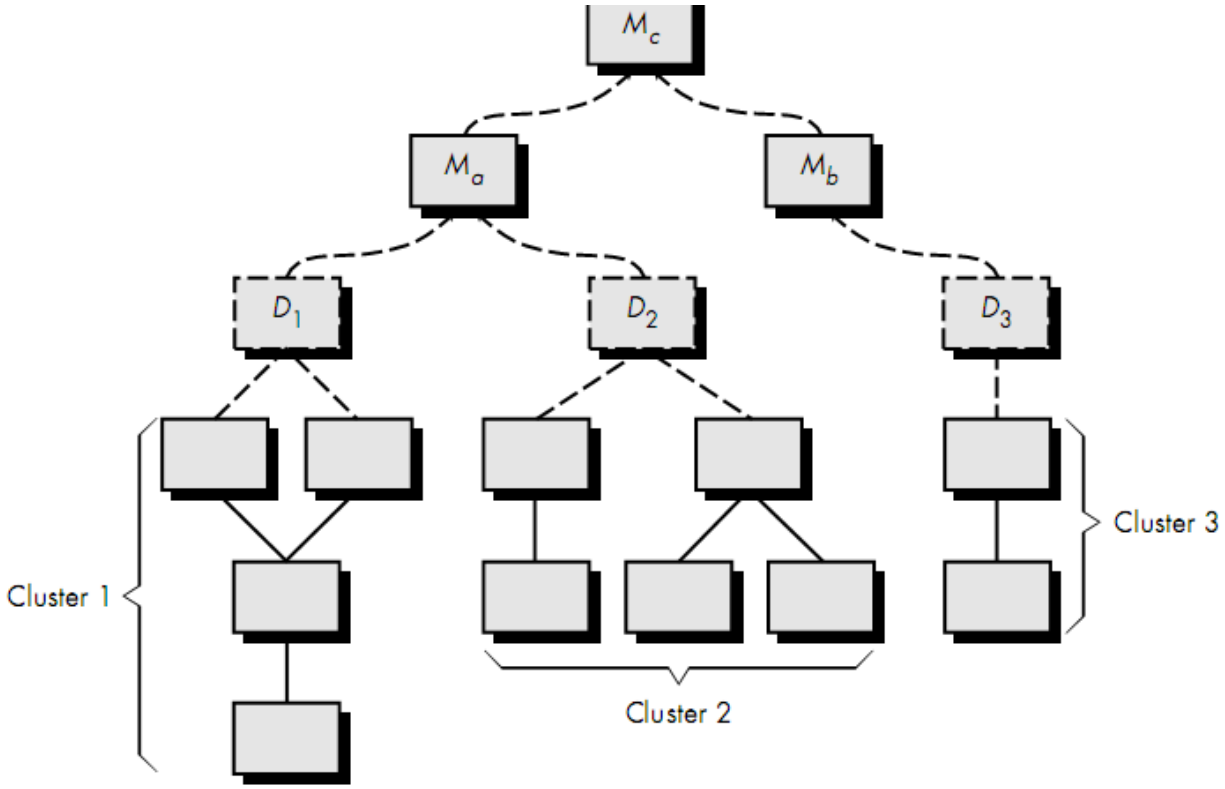
Bottom-up Integration

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 builds) that perform a specific software subfunction.
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 the Figure. Components are com-bined 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 D1 and D2 are removed and the clusters are interfaced directly to Ma. Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb. Both Ma and Mb will ultimately be integrated with component Mc, and so forth.



As integration moves upward, the need for separate test drivers lessens. In fact, if the top two levels of program structure are integrated top down, the number of drivers can be reduced substantially and integration of clusters is greatly simplified.

Sandwich integration

- Consists of a combination of both top-down and bottom-up integration
- Occurs both at the highest level modules and also at the lowest level modules
- Proceeds using functional groups of modules, with each group completed before the next
  - High and low-level modules are grouped based on the control and data processing they provide for a specific program feature
  - Integration within the group progresses in alternating steps between the high and low level modules of the group
  - When integration for a certain functional group is complete, integration and testing moves onto the next group
- Reaps the advantages of both types of integration while minimizing the need for drivers and stubs
- Requires a disciplined approach so that integration doesn't tend towards the "big bang" scenario

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 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 com-parison.
The regression test suite (the subset of tests to be executed) contains three differ-ent classes of test cases:

- A representative sample of tests that will exercise all software functions.
- Additional tests that focus on software functions that are likely to be affected by the change.
- Tests that focus on the software components that have been changed.

As integration testing proceeds, the number of regression tests can grow quite large. Therefore, the regression test suite should be designed to include only those tests that address one or more classes of errors in each of the major program functions.

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 mecha-nism 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 engi-neered 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.

The smoke test should exercise the entire system from end to end. It does not have to be exhaustive, but it should be capable of exposing major problems. The smoke test

should be thorough enough that if the build passes, you can assume that it is stable enough to be tested more thoroughly.

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 reduc-ing the likelihood of serious schedule impact when errors are uncovered.
- The quality of the end-product is improved. Because the approach is construc- tion (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 associ-ated 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

At the culmination of integration testing, software is completely assembled as a package, interfacing errors have been uncovered and corrected, and a final series of software tests—validation testing—may begin.

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 (e.g., transportability, compatibility, error recovery, maintainability).

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 bolster the support phase of the software life cycle.

Alpha and Beta Testing:

The alpha test is conducted at the developer's site by a customer. The software is used in a natural setting with the developer "looking over the shoulder" of the user and recording errors and usage problems. Alpha tests are conducted in a controlled environment.

The beta test is conducted at one or more customer sites by the end-user of the software. Unlike alpha testing, the developer is generally not present. Therefore, the beta test is a "live" application of the software in an environment that cannot be controlled by the developer. The customer records all problems (real or imagined) that are encountered during beta testing and reports these to the developer at regular intervals. As a result of problems reported during beta tests, software engineers make modifications and then prepare for release of the software product to the entire customer base.

Object-Oriented Testing

Although there are similarities between testing conventional systems and Object-Oriented systems, Object-Oriented testing has significant differences
The characteristics of Object-Oriented systems influence both testing strategy and testing methods.

- The class become the natural unit for test case design
- Implications of Object-Oriented concepts such as inheritance, encapsulation, and polymorphism pose testing challenges
- Testing the state-dependent behavior of Object-Oriented systems become important since no clear control flow in Object-Oriented programs
- Integration strategies change significantly since no obvious ‘top’ module to the system for top-down integration

- testing strategy changes
  - the concept of the ‘unit’ broadens due to encapsulation
  - integration focuses on classes and their execution
  - validation uses conventional black box methods
- test case design draws on conventional methods, but also encompasses special features

OOT Strategy

- class testing is the equivalent of unit testing
  - operations within the class are tested
  - the state behavior of the class is examined
- integration applied three different strategies
  - thread-based testing—integrates the set of classes required to respond to one input or event
  - use-based testing—integrates the set of classes required to respond to one use case

- cluster testing—integrates the set of classes required to demonstrate one collaboration

System Testing

The system software is tested as a whole. It verifies all elements mesh properly to make sure that all system functions and performance are achieved in the target environment. The focus areas are

- System functions and performance
- System reliability and recoverability (recovery test)
- System installation (installation test)
- System behavior in the special conditions (stress and load test)
- System user operations (acceptance test/alpha test)
- Hardware and software integration and collaboration
- Integration of external software and the system

Its primary purpose is to test the complete software.

1) Recovery Testing
2) Security Testing
3) Stress Testing and
4) Performance Testing

(1)Recovery tests

- Verify that the system can recover when forced to fail in various ways database recovery is particularly important
- Example: measure time to recover (MTTR)

(2) Security tests

- Verify that access protection mechanisms work make penetration cost more than value of entry
- Subject to compromise attempts
- E.G., Measure average time to break in

(3) Stress tests

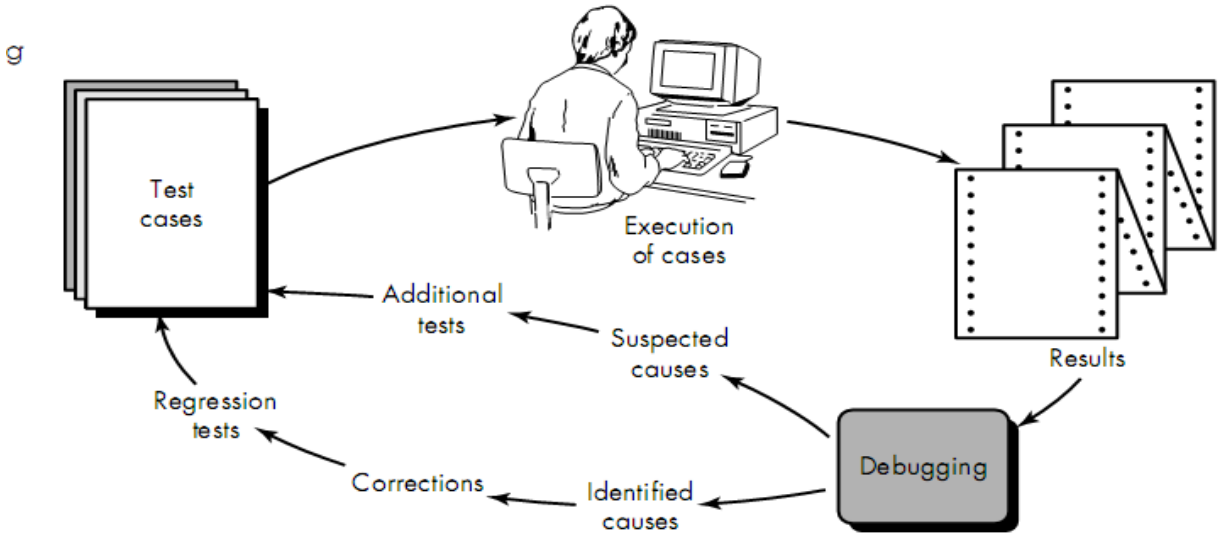
- Verify that the system can continue functioning when confronted with many simultaneous requests (abnormal situations)
- Execute the system by demanding resource in abnormal quantity, frequency, or volume (subject to extreme data & event traffic)
- Excessive interrupt, high input data rate, maximum memory, …
- How high can we go? Do we fail-soft or collapse?
- Sensitivity testing (a variation of stress testing)

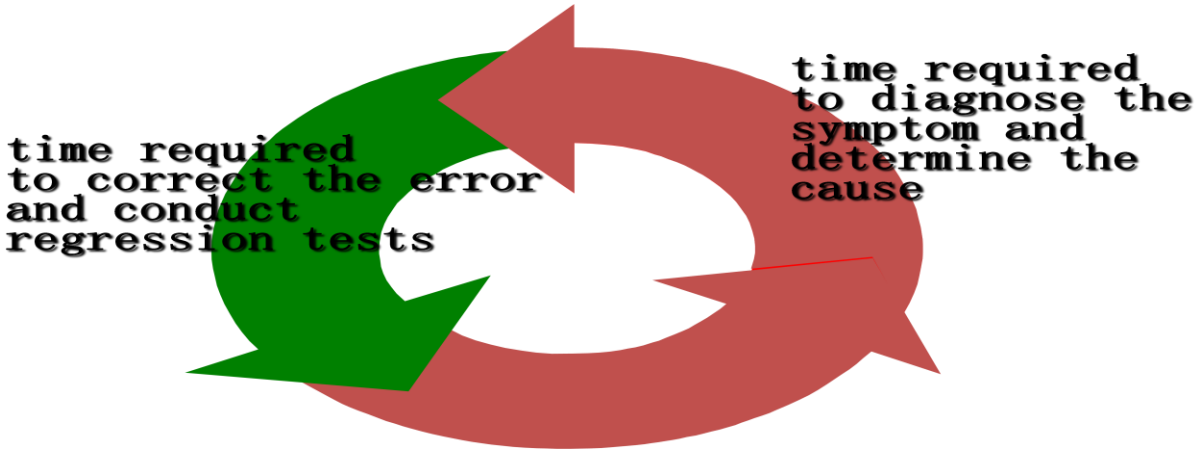
- Attempts to uncover data combinations within valid input classes that may cause instability or improper processing (performance degradation)

(4) Performance

- Is designed to test the run-time performance of software (real-time and embedded systems) within the context of an integrated system
- Measure speed, resource utilization under various circumstances.
- Is often coupled with stress testing and usually requires both hardware and software instrumentation
- Occurs throughout all steps in the testing process

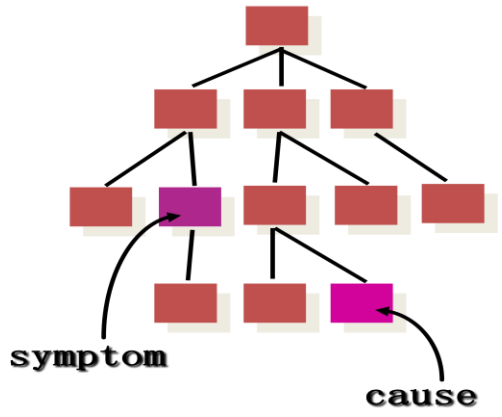
The Debugging Process





Debugging occurs as a consequence of successful testing. That is, when a test case uncovers an error, debugging is the process that results in the removal of the error. Although debugging can and should be an orderly process,

The debugging process begins with the execution of a test case. Results are assessed and a lack of correspondence between expected and actual performance is encountered. In many cases, the no corresponding data are a symptom of an underlying cause as yet hidden. The debugging process attempts to match symptom with cause, thereby leading to error correction. The debugging process will always have one of two outcomes: (1) the cause will be found and corrected, or (2) the cause will not be found. In the latter case, the per-son performing debugging may suspect a cause, design a test case to help validate that suspicion, and work toward error correction in an iterative fashion.



- symptom and cause may be geographically separated
- symptom may disappear when another problem is fixed
- cause may be due to a combination of non-errors
- cause may be due to a system or compiler error
- cause may be due to assumptions that everyone believes
- symptom may be intermittent

1. The symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled program structures exacerbate this situation.
2. The symptom may disappear (temporarily) when another error is corrected.
3. The symptom may actually be caused by nonerrors (e.g., round-off inaccuracies).
4. The symptom may be caused by human error that is not easily traced.
5. The symptom may be a result of timing problems, rather than processing problems.
6. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).
7. The symptom may be intermittent. This is particularly common in embedded systems that couple hardware and software inextricably.
8. The symptom may be due to causes that are distributed across a number of tasks running on different processors.

Debugging Approaches

1. Brute force
   Dump the memory
2. Backtracking
   Start from presence of error, go backward in the code manually
3. Cause elimination
   Cause hypothesis …

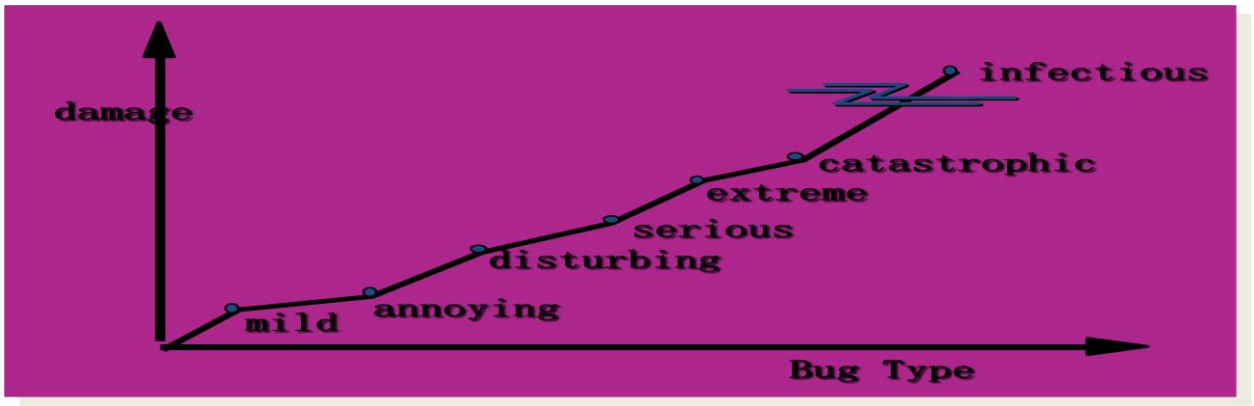
1. The brute force category of debugging is probably the most common and least efficient method for isolating the cause of a software error. We apply brute force debugging methods when all else fails. Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with WRITE statements.

2. Backtracking is a fairly common debugging approach that can be used successfully in small programs. Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the site of the cause is found. Unfortunately, as the number of source lines increases, the number of potential backward paths may become unmanageably large.

3. cause elimination—is manifested by induction or deduction and introduces the concept of binary partitioning. Data related to the error occurrence are organized to isolate potential causes. A "cause hypothesis" is devised and the aforementioned data are used to prove or disprove the hypothesis.

Each of these debugging approaches can be supplemented with debugging tools. We can apply a wide variety of debugging compilers, dynamic debugging aids ("trac-ers"), automatic test case generators, memory dumps, and cross-reference maps. However, tools are not a substitute for careful evaluation based on a complete soft-ware design document and clear source code

Consequences of Bugs



Testing Methods

Two general software testing methods:

- White-box testing: (logic-driven)
  Design tests to exercise internal structures of the software to make sure they operates according to specifications and designs
- Black-box testing: (data-driven or input/output-driven)
  Design tests to exercise each function of the software and check its errors.
- White-box and black-box testing approaches can uncover different class of errors and are complement each other

White-Box Testing

White-box testing
- Also known as glass-box testing or structural testing
- Has the knowledge of the program's structures
- A test case design method that uses the control structure of the procedural design to derive test cases
- Focus on the control structures, logical paths, logical conditions, data flows, internal data structures, and loops.
- W. Hetzel describes white-box testing as "testing in the small"

Using white-box testing methods, we can derive test cases that
- Guarantee that all independent paths within a module have been exercised at least once.
- Exercise all logical decisions on their true and false sides.
- Execute all loops at their boundaries and within their operational bounds.
- Exercise internal data structures to assure their validity.

White box testing techniques

1. Basis path testing
2. Control structure testing

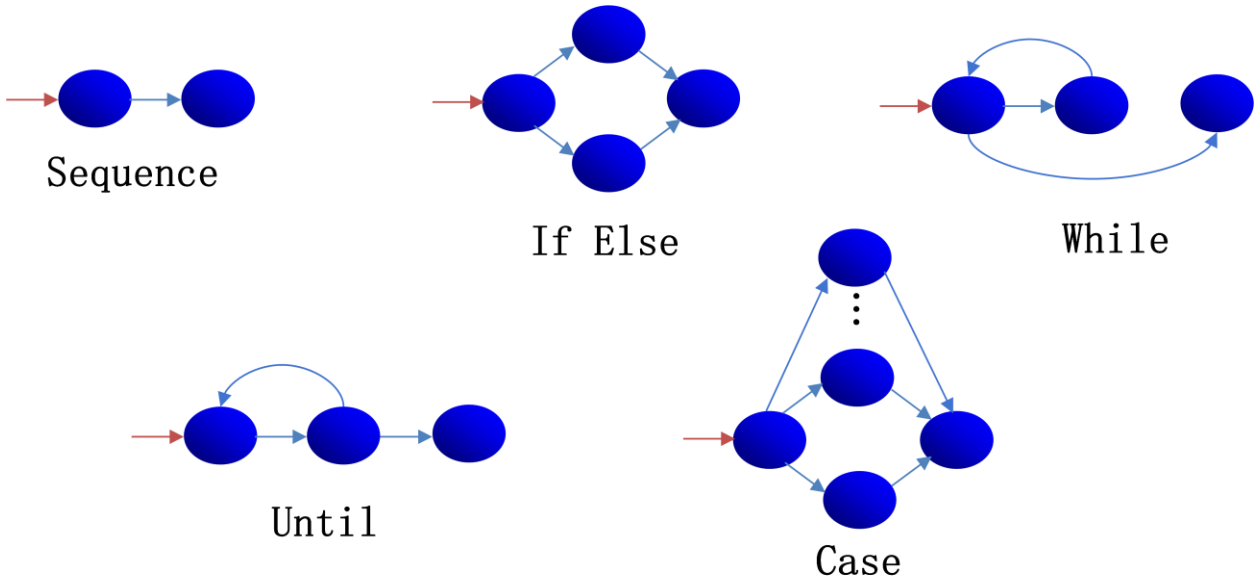
1. Basis path testing

Basic path testing (a white-box testing technique):

- First proposed by Tom McCabe.
- Can be used to derive a logical complexity measure for a procedure design.
- Used as a guide for defining a basis set of execution path.
  - Guarantee to execute every statement in the program at least one time.

➔ Flow Graph Notation
➔ Cyclomatic Complexity
➔ Deriving Test Cases

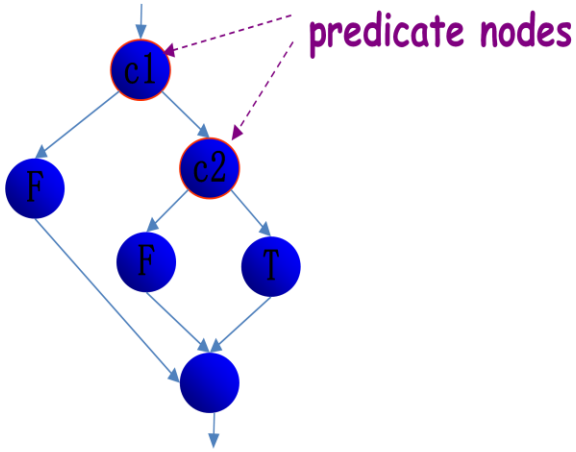
(i) Flow Graph Notation



- Flow graph notation (control flow graph)
  - Node represents one or more procedural statements.
    - A sequence of process boxes and a decision diamond can map into a single node
    - A predicate node is a node with two or more edges emanating from it
  - Edge (or link) represents flow of control
  - Region: areas bounded by edges and nodes
    - When counting regions, include the area outside the graph as a region

  - Compound condition
    - Occurs when one or more Boolean operators (OR, AND, NAND, NOR) is present in a conditional statement
    - A separate node is created for each of the conditions *C1* and *C2* in the statement *IF C1 AND C2*

```
if (c1 AND c2) then
    print T;
else
    print F;
end if;
```

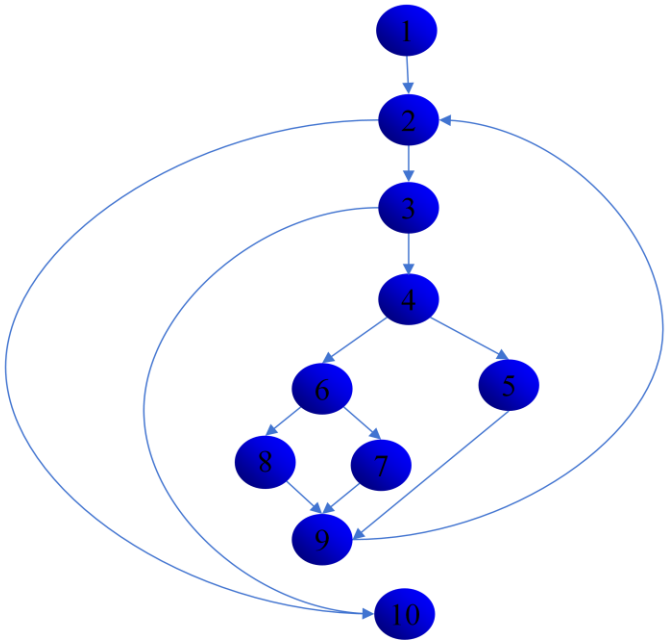


binarySearch() Example

```
public int binarySearch(int sortedArray[ ], int searchValue)
{
    int bottom = 0;
    int top = sortedArray.length - 1;
    int middle, locationOfsearchValue;
    boolean found = flase;
    locationOfsearchValue = -1;        /* the location of searchValue in the sortedArray    */
                                       /* location = -1 means that searchValue is not found */
    while ( bottom <= top && !found)
    {
        middle = (top + bottom)/2;
        if (searchValue == sortedArray[ middle ])
        {
            found = true;
            locationOfsearchValue = middle;
        }
        else if (searchValue < sortedArray[ middle ])
            top = middle - 1;
        else
            bottom = middle + 1;
    } // end while

    return locationOfsearchValue;
}
```

The Control Flow Graph (CFG) of Function binarySearch()



Cyclomatic Complexity

Cyclomatic complexity is a software metric

- provides a quantitative measure of the global complexity of a program.
- When this metric is used in the context of the basis path testing
  the value of cyclomatic complexity defines the number of independent paths in the basis set of a program
  the value of cyclomatic complexity defines an upper bound of number of tests (i.e., paths) that must be designed and exercised to guarantee coverage of all program statements

Independent path

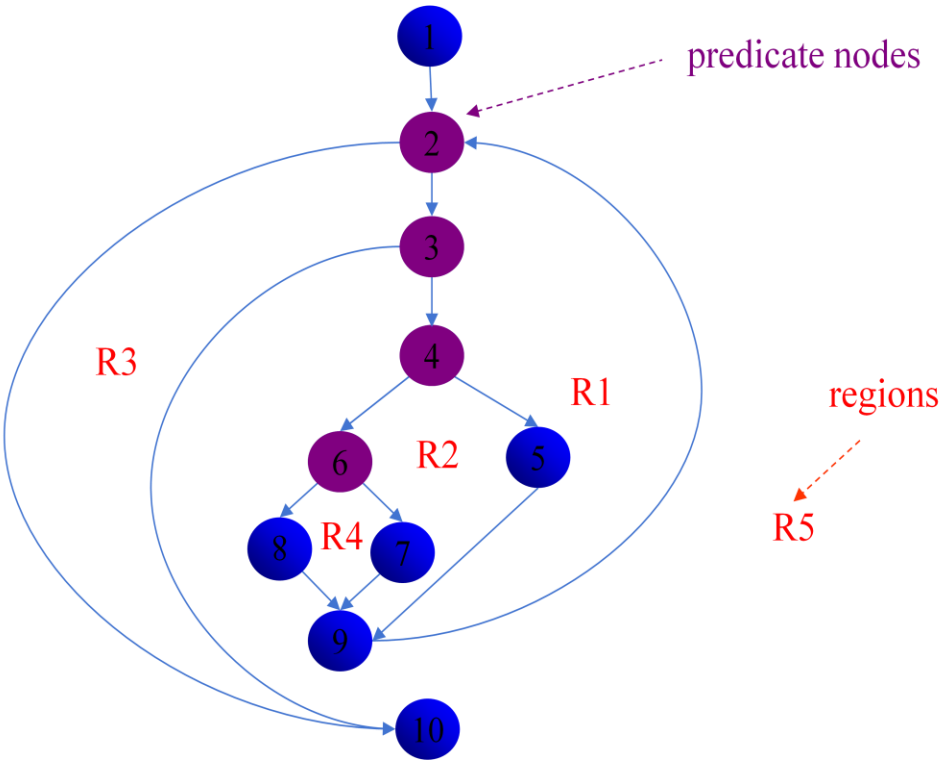
- An independent path is any path of the program that introduce at least one new set of procedural statements or a new condition
- In a flow graph, an independent path must move along at least one edge that has not been traversed before the path is defined
  Examples: consider the CFG of binarySearch()
  - Path 1: 1-2-10
  - Path 2: 1-2-3-4-6-8-9-2-10
  - Path 3: 1-2-3-4-6-8-9-2-3-10
  - Path 4: 1-2-3-4-6-8-9-2-3-4-6-8-9-2-10 (not an independent path)

Three ways to compute cyclomatic complexity:

- The number of regions of the flow graph correspond to the cyclomatic complexity.
- Cyclomatic complexity, V(G), for a flow graph G is defined as V(G) = E - N + 2
  where E is the number of flow graph edges and N is the number of flow graph nodes.
- Cyclomatic complexity, V(G) = P + 1

where P is the number of predicate nodes contained in the flow graph G.

Cyclomatic Complexity of Function binarySearch()



Deriving Basis Test Cases

The following steps can be applied to derive the basis set:

1. Using the design or code as a foundation, draw the corresponding flow graph.
2. Determine the cyclomatic complexity of the flow graph.
   - V(G) = 5 regions
   - V(G) = 13 edges – 10 nodes + 2 = 5
   - V(G) = 4 predicate nodes + 1 = 5

3. Determine a basis set of linearly independent paths.
   - Path 1: 1-2-10
   - Path 2: 1-2-3-10
   - Path 3: 1-2-3-4-5-9-2- …
   - Path 4: 1-2-3-4-6-7-9-2-…
   - Path 5: 1-2-3-4-6-8-9-2-…
4. Prepare test cases that force the execution of each path in the basis set
   - Path 1 test case:

     Inputs: sortedArray = { }, searchValue = 2
     Expected results: locationOfSearchValue = -1

     Path 2 test case: cannot be tested stand-alone!
     - Inputs: sortedArray = {2, 4, 6}, searchValue = 8
     - Expected results: locationOfSearchValue = -1

     Path 3 test case:
     - Inputs: sortedArray = {2, 4, 6, 8, 10}, searchValue = 6
     - Expected results: locationOfSearchValue = 2

     Path 4 test case:
     - Inputs: sortedArray = {2, 4, 6, 8, 10}, searchValue = 4
     - Expected results: locationOfSearchValue = 1

     Path 5 test case:
     - Inputs: sortedArray = {2, 4, 6, 8, 10}, searchValue = 10
     - Expected results: locationOfSearchValue = 4

   Each test cases is executed and compared to its expected results.
   Once all test cases have been exercised, we can be sure that all statements are executed at least once

Note: some independent paths cannot be tested stand-alone because the input data required to traverse the paths cannot be achieved

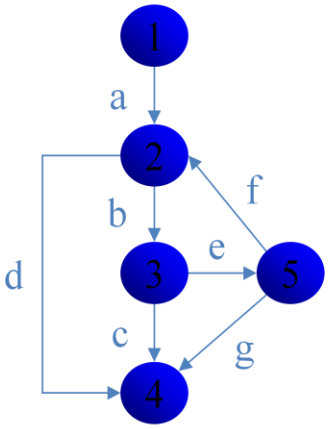
In binarySearch(), the initial value of variable *found* is FALSE, hence path 2 can only be tested as part of path 3, 4, and 5 tests

Graph Matrices

A graph matrix
- A tabular representation of a flow graph
- A square matrix with a size equal to the number of nodes on the flow graph
- Matrix entries correspond to the edges between nodes
- Adding link weight to each edge to represent
  - The connection between nodes
  - The probability of the edge to be executed
  - The resource (e.g., processing time or memory) required for traversing the edge

A connection matrix

- A graph matrix with the link weight is 1 (representing a connection exists) or 0 (representing a connection does not exist)
- Each row of the matrix with two or more entries represents a predicate node
- Provide another method for computing the cyclomatic complexity of a flow graph



Flow Graph

Node \ Connected to node

	1	2	3	4	5
1		a			
2			b	d	
3				c	e
4					
5		f		g	

Graph Matrix

	1	2	3	4	5
1		1			
2			1	1	
3				1	1
4					
5		1		1	

Connection Matrix

Connections

1 - 1 = 0

2 - 1 = 1

2 - 1 = 1

2 - 1 = 1

---

3 + 1 = 4

Cyclomatic complexity

Product metrics

- Product metrics for computer software helps us to assess quality.
- Measure

-- Provides a quantitative indication of the extent, amount, dimension, capacity or size of some attribute of a product or process

- Metric(IEEE 93 definition)

-- A quantitative measure of the degree to which a system, component or process possess a given attribute

- Indicator

-- A metric or a combination of metrics that provide insight into the software process, a software project or a product itself

- **Product metrics for the Analysis model**

❖ Function point Metric

- First proposed by Albrecht
- Measures the functionality delivered by the system
- FP computed from the following parameters

1) Number of external inputs(EIS)
2) Number external outputs(EOS)
3) Number of external Inquiries(EQS)
4) Number of Internal Logical Files(ILF)
5) Number of external interface files(EIFS)

Each parameter is classified as simple, average or complex and weights are assigned as follows

Information Domain	Count	Simple	avg	Complex
EIS		3	4	6
EOS		4	5	7
EQS		3	4	6
ILFS		7	10	15
EIFS		5	7	10

FP=Count total *[0.65+0.01*E(Fi)]

Metrics for Design Model

- DSQI(Design Structure Quality Index)
- US air force has designed the DSQI
- Compute s1 to s7 from data and architectural design
- S1:Total number of modules
- S2:Number of modules whose correct function depends on the data input
- S3:Number of modules whose function depends on prior processing
- S4:Number of data base items
- S5:Number of unique database items
- S6: Number of database segments
- S7:Number of modules with single entry and exit
- Calculate D1 to D6 from s1 to s7 as follows:
- D1=1 if standard design is followed otherwise D1=0
- D2(module independence)=(1-(s2/s1))
- D3(module not depending on prior processing)=(1-(s3/s1))
- D4(Data base size)=(1-(s5/s4))
- D5(Database compartmentalization)=(1-(s6/s4)
- D6(Module entry/exit characteristics)=(1-(s7/s1))
- DSQI=sigma of WiDi
- i=1 to 6,Wi is weight assigned to Di
- If sigma of wi is 1 then all weights are equal to 0.167
- DSQI of present design be compared with past DSQI. If DSQI is significantly lower than the average, further design work and review are indicated

METRIC FOR SOURCE CODE

- HSS(Halstead Software science)
- Primitive measure that may be derived after the code is generated or estimated once design is complete
- n_1 = the number of distinct operators that appear in a program
- n_2 = the number of distinct operands that appear in a program
- N_1 = the total number of operator occurrences.
- N_2 = the total number of operand occurrence.
- Overall program length N can be computed:
- $N = n_1 \log2\ n_1 + n_2 \log2\ n_2$
- $V = N \log_2 (n_1 + n_2)$

METRIC FOR TESTING

- n_1 = the number of distinct operators that appear in a program
- n_2 = the number of distinct operands that appear in a program
- N_1 = the total number of operator occurrences.
- N_2 = the total number of operand occurrence.
- Program Level and Effort
- $PL = 1/[(n_1 / 2) \times (N_2 / n_2\ l)]$
- $e = V/PL$

METRICS FOR MAINTENANCE

- M_t = the number of modules in the current release
- F_c = the number of modules in the current release that have been changed
- F_a = the number of modules in the current release that have been added.
- F_d = the number of modules from the preceding release that were deleted in the current release
- The Software Maturity Index, SMI, is defined as:
- $SMI = [M_{t-(}F_c + F_{a+}F_{d)/}M_t]$