

Software Engineering

Lecture 10

Testing Strategies

Strategic approach to software testing

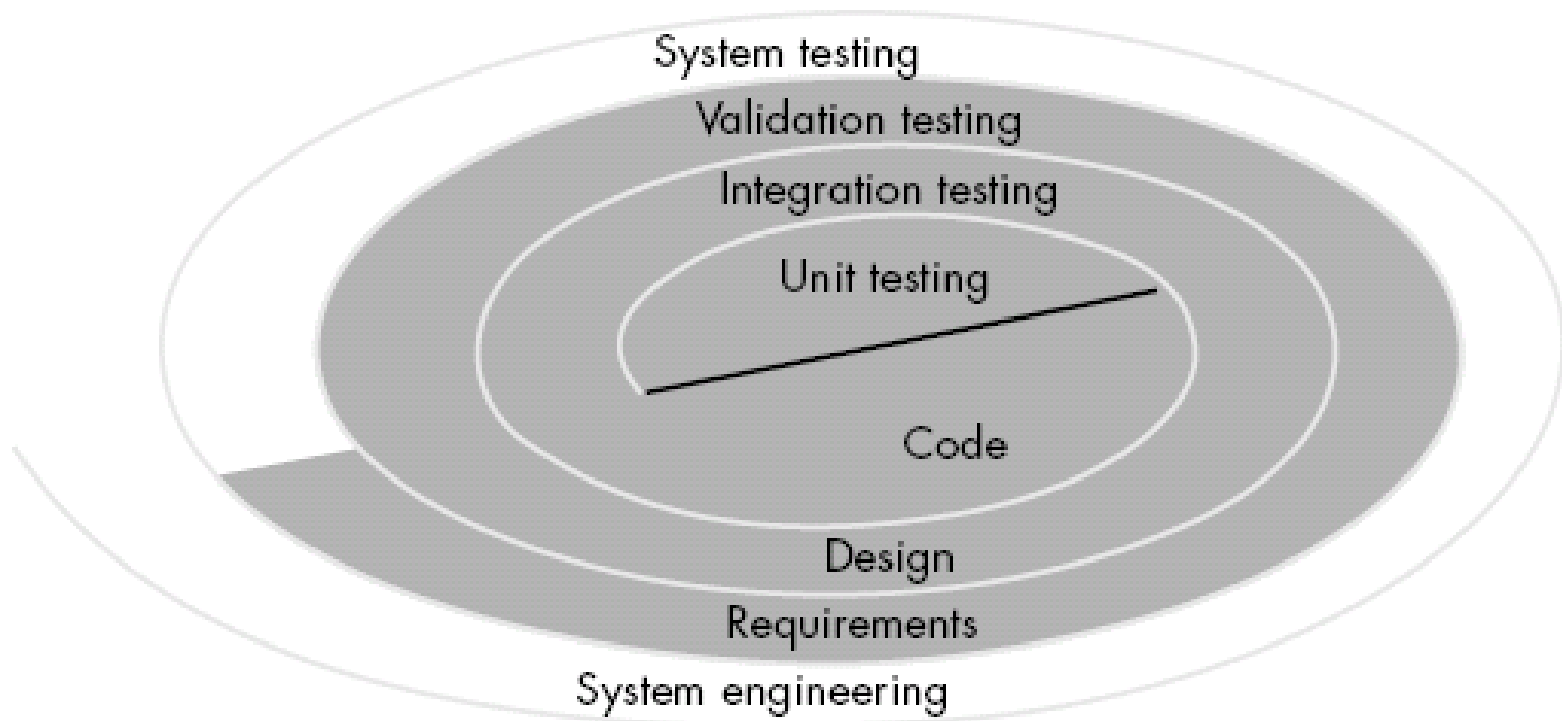
- Generic characteristics of strategic software testing:
 - To perform effective testing, a software team should conduct effective *formal technical reviews*. By doing this, many errors will be eliminated before testing start.
 - Testing begins at the *component level* and works "*outward*" toward the integration of the entire computer-based system.
 - Different *testing techniques* are appropriate at different points in *time*.
 - Testing is conducted by the developer of the software and (for large projects) an independent test group.
 - *Testing and debugging are different activities*, but debugging must be accommodated in any testing strategy.


Verification and Validation

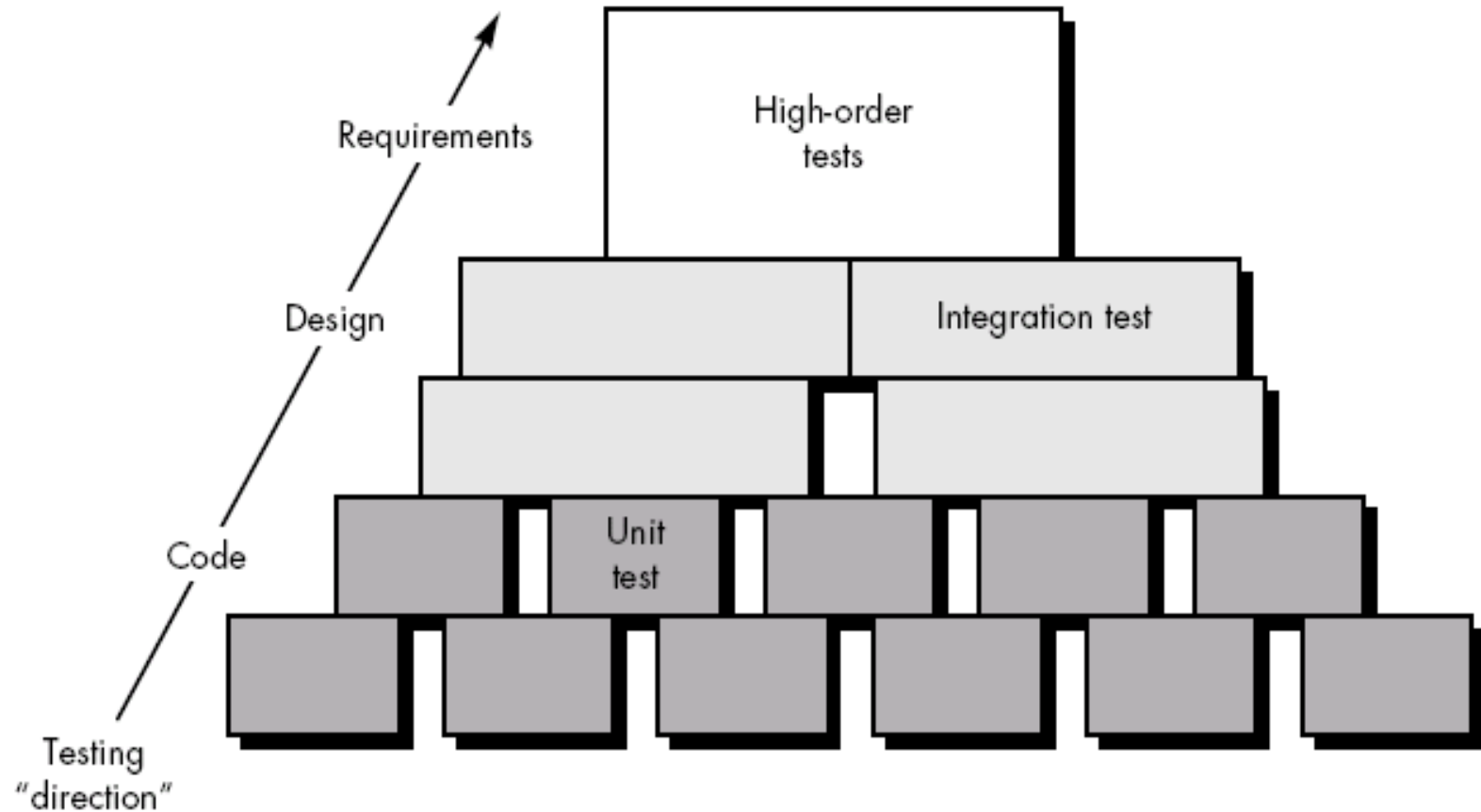
- Testing is one element of a broader topic that is often referred to as *verification and validation (V&V)*.
- *Verification* refers to the set of activities that ensure that software correctly implements a specific function.
- *Validation* refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements.
- State another way:
 - *Verification*: "Are we building the product right?"
 - *Validation*: "Are we building the right product?"
- The definition of V&V encompasses many of the activities that are similar to *software quality assurance (SQA)*.

- V&V encompasses a wide array of SQA activities that include
 - Formal technical reviews,
 - documentation review,
 - quality and configuration audits,
 - performance monitoring,
 - simulation,
 - feasibility study,
 - database review,
 - algorithm analysis,
 - development testing,
 - qualification testing, and installation testing
- Testing does provide the last bastion from which quality can be assessed and, more pragmatically, errors can be uncovered.
- Quality is not measure only by no. of error but it is also measure on application methods, process model, tool, formal technical review, etc will lead to quality, that is confirmed during testing.


Software Testing Strategy for conventional software architecture



- 
- A *Software process & strategy for software testing* may also be viewed in the context of the *spiral*.
 - *Unit testing* begins at the vortex of the spiral and concentrates on each unit (i.e., component) of the software.
 - Testing progresses by moving outward along the spiral to *integration testing*, where the focus is on design and the construction.
 - Another turn outward on the spiral, we encounter *validation testing*, where requirements established as part of software requirements analysis are validated against the software.
 - Finally, we arrive at *system testing*, where the software and other system elements are tested as a whole.




- Software process from a procedural point of view; a series of four steps that are implemented sequentially.

- 
- Initially, tests focus on each component individually, ensuring that it functions properly as a unit.
 - Unit testing makes heavy use of white-box testing techniques, exercising specific paths in a module's control structure.
 - Integration testing addresses the issues associated with the dual problems of verification and program construction.
 - Black-box test case design techniques are the most prevalent during integration.
 - Now, validation testing provides final assurance that software meets all functional, behavioral, and performance requirements.
 - Black-box testing techniques are used exclusively during validation.
 - once validated, must be combined with other system elements (e.g., hardware, people, databases). System testing verifies that all elements mesh properly and that overall system function / performance is achieved.

Criteria for Completion of Testing

- There is no definitive answer to state that “we have done with testing”.
- One response to the question is: "You're never done testing, the burden simply shifts from you (the software engineer) to your customer." Every time the customer/user executes a computer program, the program is being tested.
- Another response is: "You're done testing when you run out of time (deadline to deliver product to customer) or you run out of money (*spend so much money on testing*).

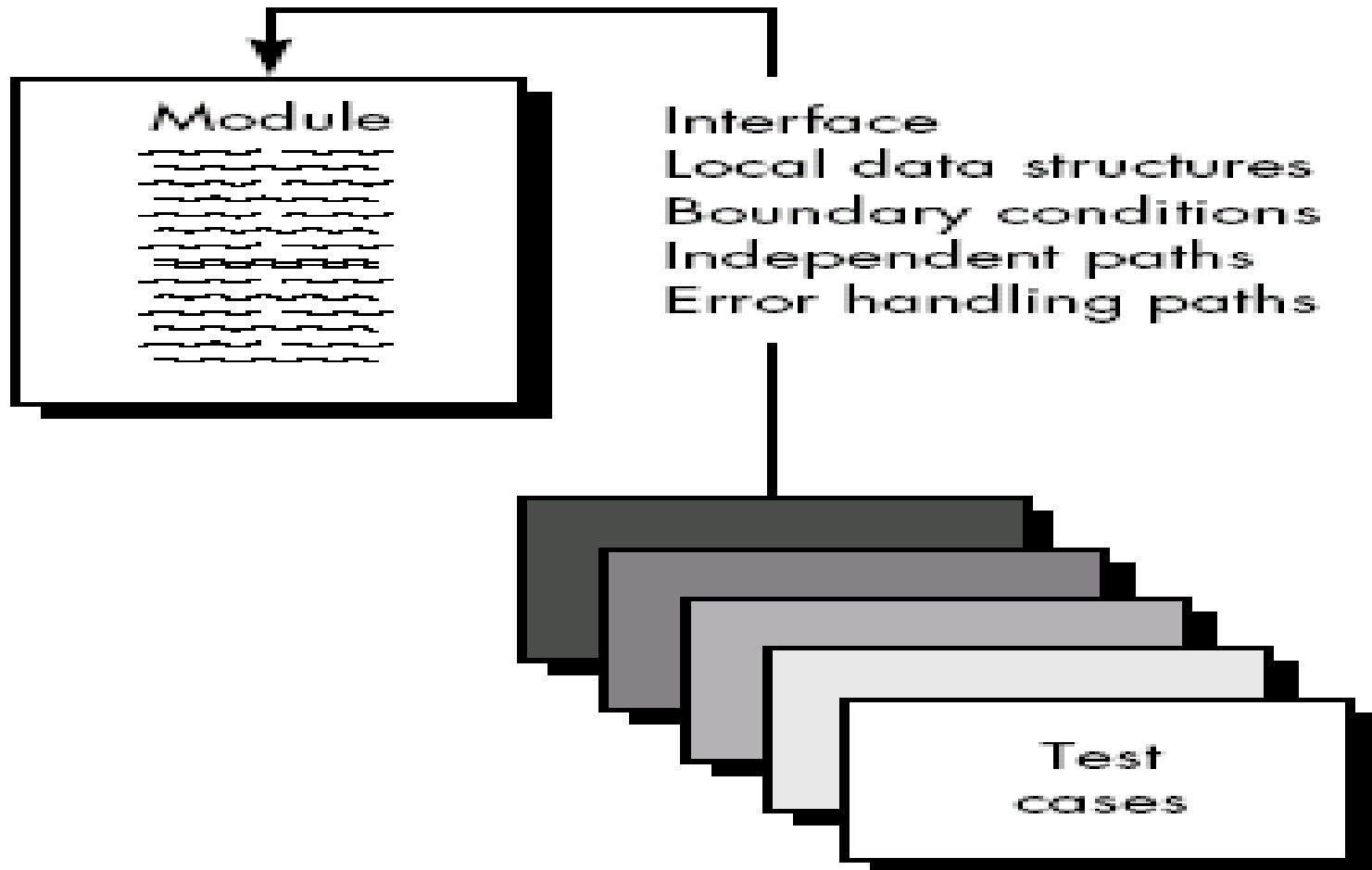
- 
- But few practitioners would argue with these responses, a software engineer needs more rigorous criteria for determining when sufficient testing has been conducted.
 - Response that is based on statistical criteria: "No, we cannot be absolutely predict that the software will never fail, but relative to a theoretically sound and experimentally validated statistical model, we have done sufficient testing to say with 95 percent confidence that program will not fail."



Unit testing strategies for conventional software


- Focuses verification effort on the smallest unit of software design – 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.
- Unit test is white-box oriented, and the step can be conducted in parallel for multiple components.
- Unit test consists of
 - Unit Test Considerations
 - Unit Test Procedures

Unit Test Considerations



Contd.

- *Module interface* - information properly flows into and out of the program unit under test.
- *local data structure* - data stored temporarily maintains its integrity.
- *Boundary conditions* - module operates properly at boundaries established to limit or restrict processing
- Independent paths - all statements in a module have been executed at least once.
- And finally, all *error handling paths* are tested.

- 
- *Module interface* are required before any other test is initiated because If data do not enter and exit properly, all other tests are debatable.
 - In addition, *local data structures* should be exercised and the local impact on global data should be discover during unit testing.
 - Selective testing of *execution paths* is an essential task during the unit test. Test cases should be designed to uncover errors due to
 - Computations,
 - Incorrect comparisons, or
 - Improper control flow
 - *Basis path* and loop testing are effective techniques for uncovering a broad array of *path errors*.

Errors are commonly found during unit testing

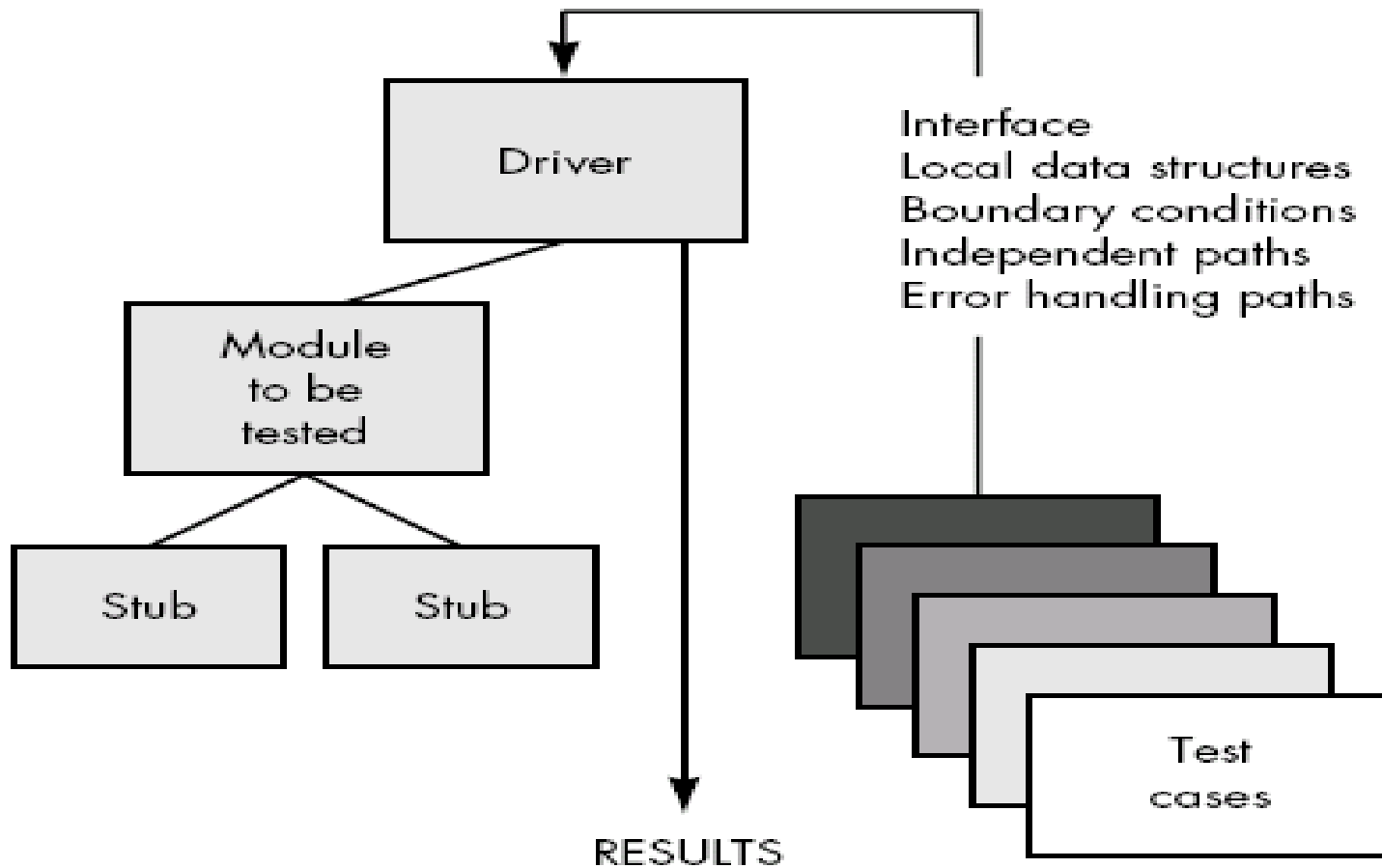
- More common errors in computation are
 - misunderstood or incorrect arithmetic precedence
 - mixed mode operations,
 - incorrect initialization,
 - precision inaccuracy,
 - incorrect symbolic representation of an expression.
- Comparison and control flow are closely coupled to one another
 - Comparison of different data types,
 - Incorrect logical operators or precedence,
 - Incorrect comparison of variables
 - Improper or nonexistent loop termination,
 - Failure to exit when divergent iteration is encountered
 - improperly modified loop variables.

- Potential errors that should be tested when error handling is evaluated are
 - Error description is unintelligible.
 - Error noted does not correspond to error encountered.
 - Error condition causes system intervention prior to error handling.
 - Exception-condition processing is incorrect.
 - Error description does not provide enough information to assist in the location of the cause of the error.
- Software often fails at its boundaries. That is, errors often occur when the n th element of an n -dimensional array is processed or when the maximum or minimum allowable value is encountered.
- So BVA test is always be a last task for unit test.
- Test cases that exercise data structure, control flow, and data values just below, at, and just above maxima and minima are very likely to uncover errors.


Unit Test Procedures

- Perform before coding or after source code has been generated.
- A review of design information provides guidance for establishing test cases. Each test case should be coupled with a set of expected results.
- Because a component is not a stand-alone program, driver and/or stub software must be developed for each unit test.
- In most applications a driver 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.
- 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.
- *Stubs* serve to replace modules that are subordinate the component to be tested.

Unit Test Procedures



Unit Test Environment

- 
- Drivers and stubs represent overhead. That is, both are software that must be written but that is not delivered with the final software product.
 - In such cases, complete testing can be postponed until the integration test step
 - Unit testing is simplified when a component with high cohesion is designed.
 - When only one function is addressed by a component, the number of test cases is reduced and errors can be more easily predicted and uncovered.



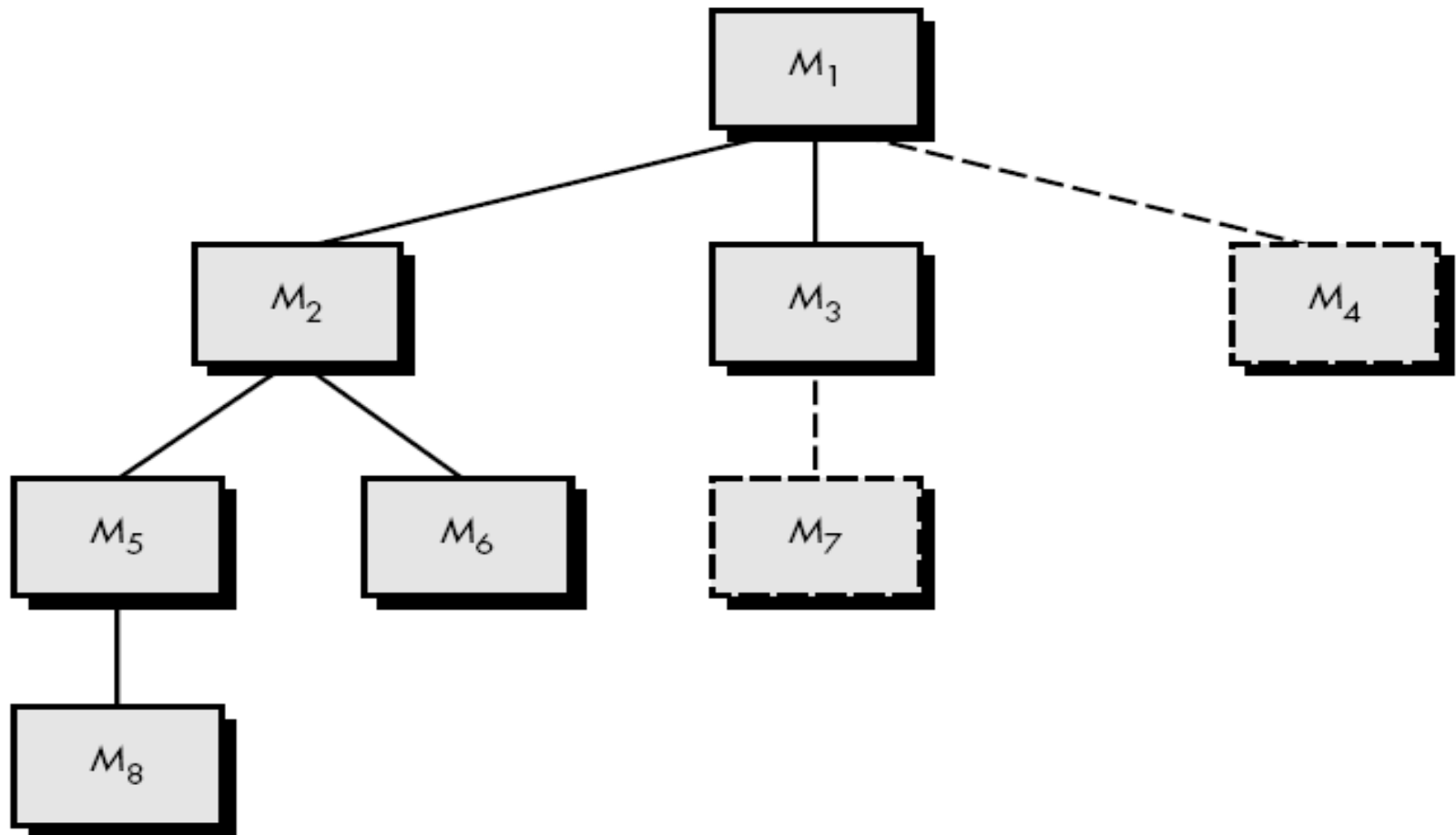
Integration testing


- 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.
- There is often a tendency to attempt non-incremental integration; that is, to construct the program using a "big bang" approach.
- A set of errors is encountered. Correction is difficult because isolation of causes is complicated 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.
- Incremental integration is the exact opposite of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct;

Top-down Integration

- *Top-down integration testing is an incremental approach to construction of program structure.*
- Modules 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 M6 would be integrated
 - The central and right hand control paths are built.

Top down integration



- 
- *Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally.*
 - Step would be:
 - components M2, M3, and M4 would be integrated first
 - next control level, M5, M6, and so on follows.



Top-down Integration process 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.

Problem occur in top-down integration

- Logistic problems can arise
- most common problems occurs when processing at low levels in the hierarchy is required to adequately test upper levels.
- No significant data can flow upward in the program structure due to stubs replace low level modules at the beginning of top-down testing. In this case, Tester will have 3 choice
 - Delay many tests until stubs are replaced with actual modules
 - develop stubs that perform limited functions that simulate the actual module
 - integrate the software from the bottom of the hierarchy upward



Bottom-up Integration

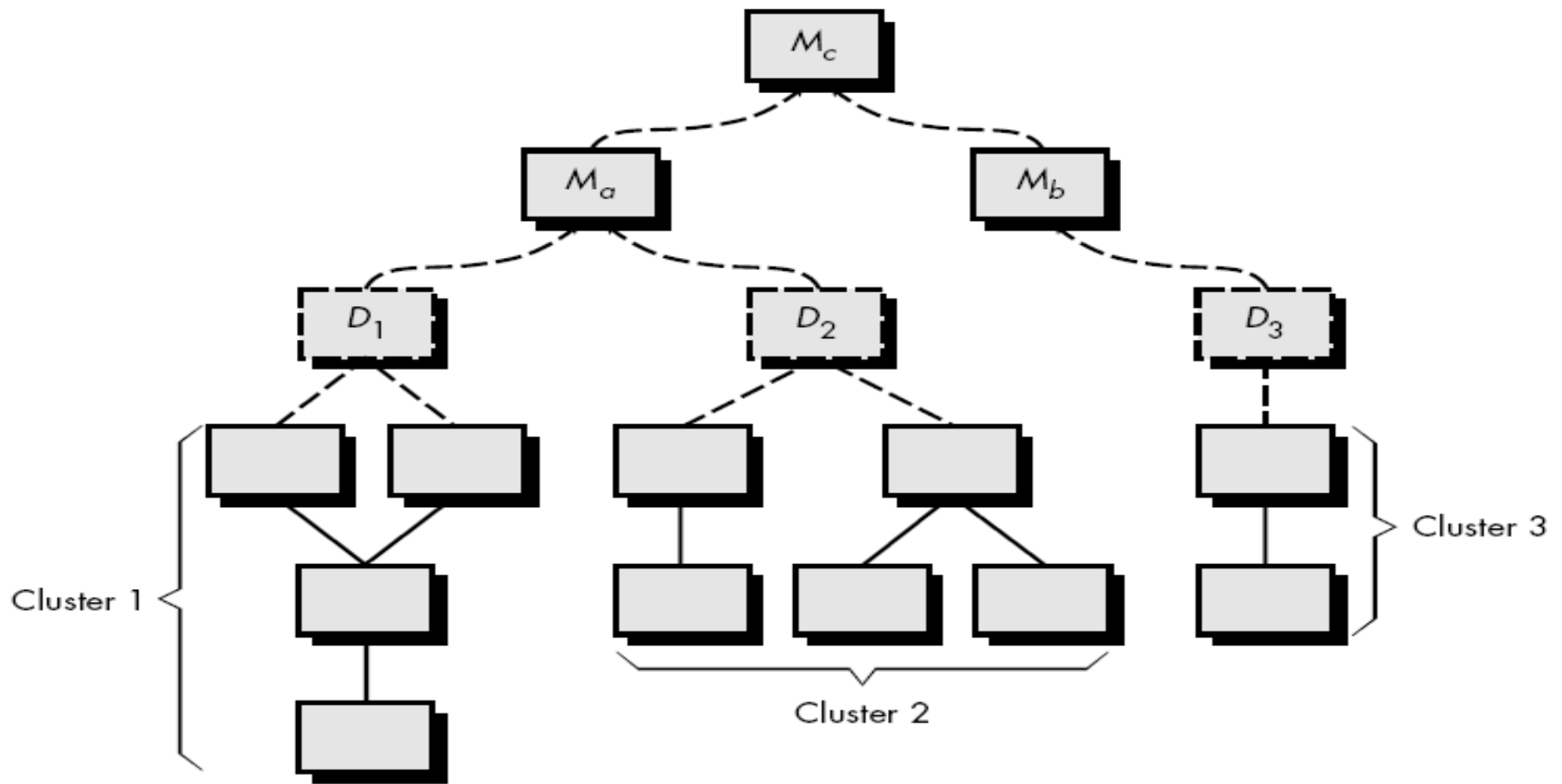
- *Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure)*
- Because components are integrated from the bottom up, processing required for components subordinate to a given level is always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



Bottom up integration process steps

- Low-level components are combined into clusters (sometimes called *builds*) that perform a specific software sub function.
- A driver (a control program for testing) is written to coordinate test case input and output.
- The cluster is tested.
- Drivers are removed and clusters are combined moving upward in the program structure.

Bottom up integration



Example

- Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver.
- 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.

Regression Testing

- Each time a new module is added as part of integration testing
 - New data flow paths are established
 - New I/O may occur
 - New control logic is invoked
- Due to these changes, may cause problems with functions that previously worked flawlessly.
- *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.
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.

Contd.

- 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.
- Regression testing contains 3 diff. 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.



Contd.

- As integration testing proceeds, the number of regression tests can grow quite large.
- 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.
- It is impractical and inefficient to re-execute every test for every program function once a change has occurred.

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.

Smoke testing approach activities

- 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.
- 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 stopper” errors that have the highest likelihood of throwing the software project behind schedule.
- The build is integrated with other builds and the entire product is smoke tested daily.
 - The integration approach may be top down or bottom up.

Smoke Testing benefits

- *Integration risk is minimized.*
 - Smoke tests are conducted daily, incompatibilities and other show-stopper errors are uncovered early
- *The quality of the end-product is improved.*
 - Smoke testing is likely to uncover both functional errors and architectural and component-level design defects. At the end, better product quality will result.
- *Error diagnosis and correction are simplified.*
 - Software that has just been added to the build(s) is a probable cause of a newly discovered error.
- *Progress is easier to assess.*
 - Frequent tests give both managers and practitioners a realistic assessment of integration testing progress.

What is a critical module and why should we identify it?

- As integration testing is conducted, the tester should identify *critical modules*.
- A critical module has one or more of the following characteristics:
 - ☐ Addresses several software requirements,
 - ☐ Has a high level of control (Program structure)
 - ☐ Is complex or error prone
 - ☐ Has definite performance requirements.
- Critical modules should be tested as early as is possible. In addition, regression tests should focus on critical module function.



Integration Test Documentation

- An overall plan for integration of the software and a description of specific tests are documented in a *Test Specification*
- It contains a test plan, and a test procedure, is a work product of the software process, and becomes part of the software configuration.
- The test plan describes the overall strategy for integration.
- Testing is divided into phases and builds that address specific functional and behavioral characteristics of the software.
- Integration testing might be divided into the following test phases:
 - User interaction
 - Data manipulation and analysis
 - Display processing and generation
 - Database management



Contd.

- Therefore, groups of modules are created to correspond to each phase.
- The following criteria and corresponding tests are applied for all test phases:
- **Interface integrity**- Internal and external interfaces are tested as each module
- **Functional validity** - Tests designed to uncover functional errors are conducted
- **Information content** - associated with local or global data structures are conducted
- **Performance** - to verify performance

Contd.

- A schedule for integration and related topics is also discussed as part of the test plan.
- Start and end dates for each phase are established
- A brief description of overhead software (stubs and drivers) concentrates on characteristics that might require special effort.
- Finally, test environment and resources are described.
- The order of integration and corresponding tests at each integration step are described.
- A listing of all test cases and expected results is also included.
- A history of actual test results, problems is recorded in the *Test Specification*.

Validation Testing

- *Validation testing* succeeds when software functions in a manner that can be *reasonably expected by the customer*.
- Like all other testing steps, validation tries to uncover errors, but the focus is at the requirements level— on things that will be immediately apparent to the end-user.
- Reasonable expectations are defined in the *Software Requirements Specification*— a document that describes all user-visible attributes of the software.
- Validation testing comprises of
 - Validation Test criteria
 - Configuration review
 - Alpha & Beta Testing

Validation Test criteria

- It is achieved through a series of tests that demonstrate agreement 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 *agreement 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,
 - ☐ other requirements are met
- After each validation test case has been conducted, one of two possible conditions exist:
 1. The function or performance characteristics conform to specification and are accepted
 2. A deviation from specification is uncovered and a deficiency list is created



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 the support phase of the software life cycle.
- The configuration review, sometimes called an *audit*.

Alpha and Beta Testing

- When custom software is built for one customer, a series of *acceptance tests* are conducted to enable the customer to validate all requirements.
- Conducted by the end-user rather than software engineers, an acceptance test can range from an informal "test drive" to a planned and systematically executed series of tests.
- Most software product builders use a process called alpha and beta testing to uncover errors that only the end-user seems able to find.



Alpha 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.



Beta testing

- The *beta test* is conducted at one or more customer sites by the end-user of the software.
- 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.

System Testing

- System testing is actually a series of different tests whose primary purpose is to fully exercise the computer-based system.
- Although each test has a different purpose, all work to verify that system elements have been properly integrated and perform allocated functions.
- Types of system tests are:
 - Recovery Testing
 - Security Testing
 - Stress Testing
 - Performance Testing



Recovery Testing

- *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 by the system itself), reinitialization, checkpointing mechanisms, data recovery, and restart are evaluated for correctness.
- If recovery requires human intervention, that is mean-time-to-repair (MTTR) is evaluated to determine whether it is within acceptable limits.



Security Testing

- *Security testing* attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper break through .
- During security testing, the tester plays the role(s) of the individual who desires to break through the system.
- 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.
- The tester may attempt to acquire passwords through externally, may attack the system with custom software designed to breakdown any defenses that have been constructed; may browse through insecure data; may purposely cause system errors.

Stress Testing

- *Stress testing* executes a system in a manner that demands resources in abnormal quantity, frequency, or volume.

For example,

1. special tests may be designed that generate ten interrupts per second
 2. Input data rates may be increased by an order of magnitude to determine how input functions will respond
 3. test cases that require maximum memory or other resources are executed
 4. test cases that may cause excessive hunting for disk-resident data are created
- A variation of stress testing is a technique called *sensitivity testing*



Performance Testing

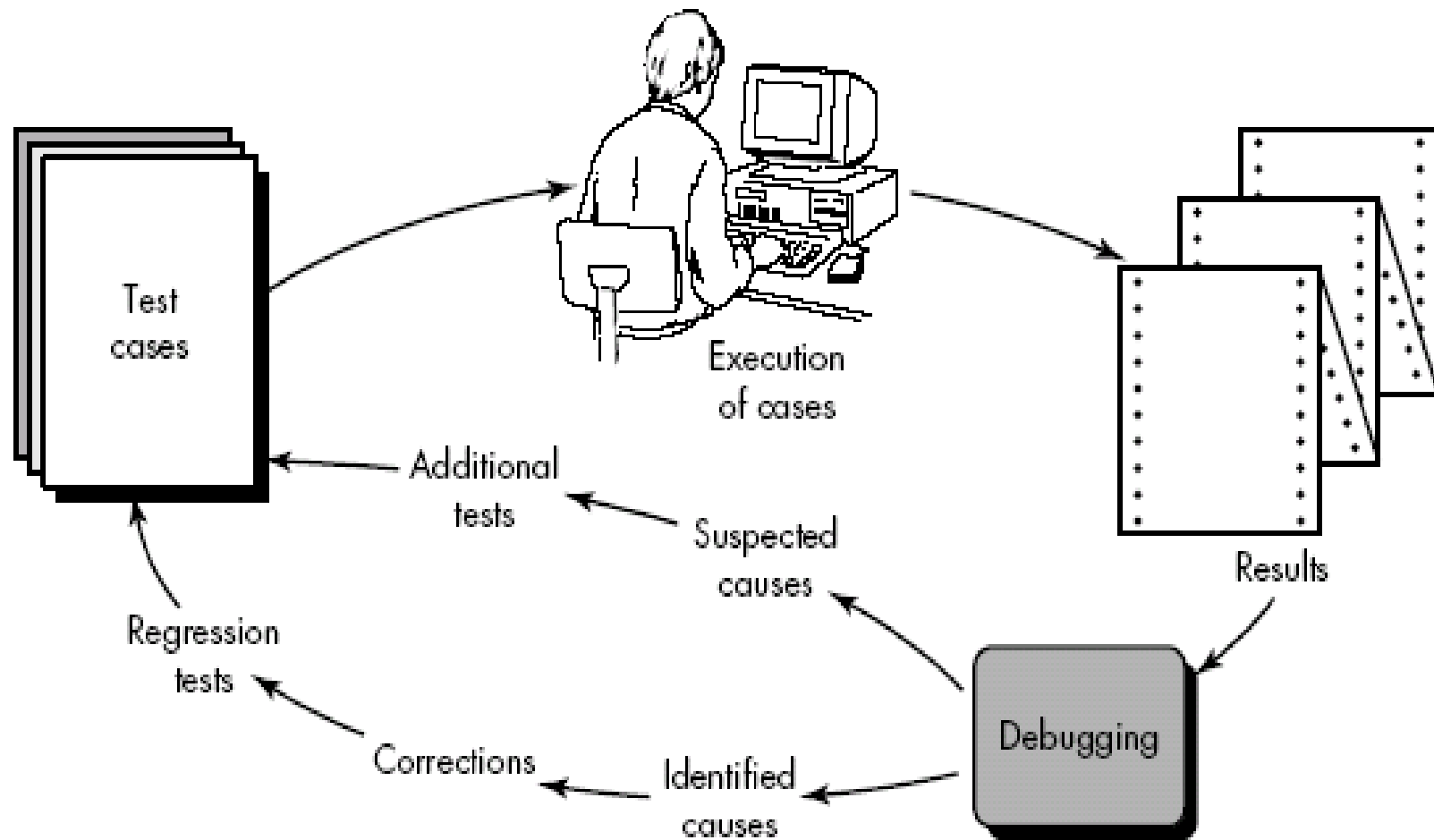
- Performance testing occurs throughout all steps in the testing process.
- Even at the unit level, the performance of an individual module may be assessed as white-box tests are conducted.
- Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation
- It is often necessary to measure resource utilization (e.g., processor cycles).



THE ART OF DEBUGGING

- Debugging is the process that results in the **removal of the error**.
- Although debugging can and should be an orderly process, it is still very much an art.
- Debugging is not testing but always occurs as a **consequence of testing**.

Debugging Process




Debugging Process

- Results are examined and a lack of correspondence between expected and actual performance is encountered (due to cause of error).
- Debugging process attempts to match symptom with cause, thereby leading to error correction.
- One of two outcomes always comes from debugging process:
 - The cause will be found and corrected,
 - The cause will not be found.
- The person performing debugging may *suspect a cause*, design a test case to help validate that doubt, and work toward error correction in an iterative fashion.

Why is debugging so difficult?

1. The symptom may disappear (temporarily) when another error is corrected.
2. The symptom may actually be caused by non-errors (e.g., round-off inaccuracies).
3. The symptom may be caused by **human error** that is not easily traced (e.g. **wrong input**, wrongly configure the system)
4. The symptom may be a result of timing problems, rather than processing problems.(e.g. **taking so much time** to display result).
5. It may be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).

- 
6. The symptom may be intermittent (connection irregular or broken). This is particularly common in embedded systems that couple hardware and software
 7. The symptom may be due to causes that are distributed across a number of tasks running on different processors

As the consequences of an error increase, the amount of pressure to find the cause also increases. Often, pressure sometimes forces a software developer to fix one error and at the same time introduce two more.

Debugging Approaches or strategies

- Debugging has one overriding objective: to find and correct the cause of a software error.
- Three categories for debugging approaches
 - Brute force
 - Backtracking
 - Cause elimination

Brute Force:

- probably the most common and **least efficient** method for isolating the cause of a software error.
- 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 or PRINT statements
- It more frequently leads to **wasted effort and time**.

Backtracking:

- common debugging approach that can be used successfully **in small programs**.
- Beginning at the site where a symptom has been open, **the source code is traced backward** (manually) until the site of the cause is found.

Cause elimination

- Is cleared by induction or deduction and introduces the concept of **binary partitioning** (i.e. valid and invalid).
- A list of all possible causes is developed and tests are conducted to eliminate each.

Correcting the error

- The correction of a bug can introduce other errors and therefore do more harm than good.

Questions that every software engineer should ask before making the "correction" that removes the cause of a bug:

- **Is the cause of the bug reproduced in another part of the program?** (i.e. cause of bug is logical pattern)
- **What "next bug" might be introduced by the fix I'm about to make?** (i.e. cause of bug can be in logic or structure or design).
- **What could we have done to prevent this kind of bug previously?** (i.e. same kind of bug might generated early so developer can go through the steps)