Software Testing Strategies

What is Testing Strategy?

- Designing effective test cases is important, but so is the strategy you use to execute them. Should you develop a formal plan for your tests? Should you test the entire program as a whole or run tests only on a small part of it? Should you rerun tests you've already conducted as you add new components to a large system? When should you involve the customer? These and many other questions are answered when you develop a software testing strategy.

A Strategic Approach to Software Testing

- 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.

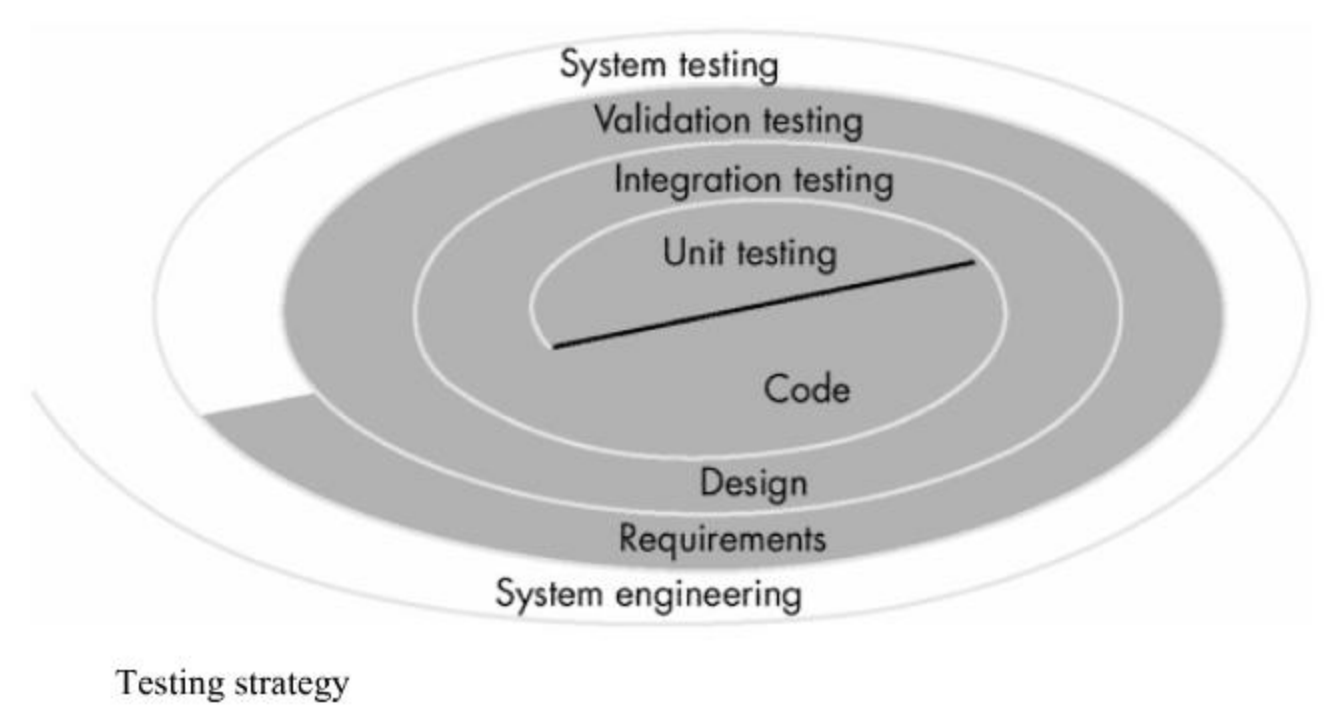
Verification and Validation

- **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.
- **Verification: "Are we building the product right?“**
- **Validation: "Are we building the right product?"**

Organizing for Software Testing

- The software developer is always responsible for testing the individual units (components) of the program, ensuring that each performs the function for which it was designed.
- In many cases, the developer also conducts integration testing.
- Only after the software architecture is complete does an independent test group become involved.
- The role of an independent test group (ITG) is to remove the inherent problems associated with letting the builder test the thing that has been built. Independent testing removes the conflict of interest that may otherwise be present.

A Software Testing Strategy



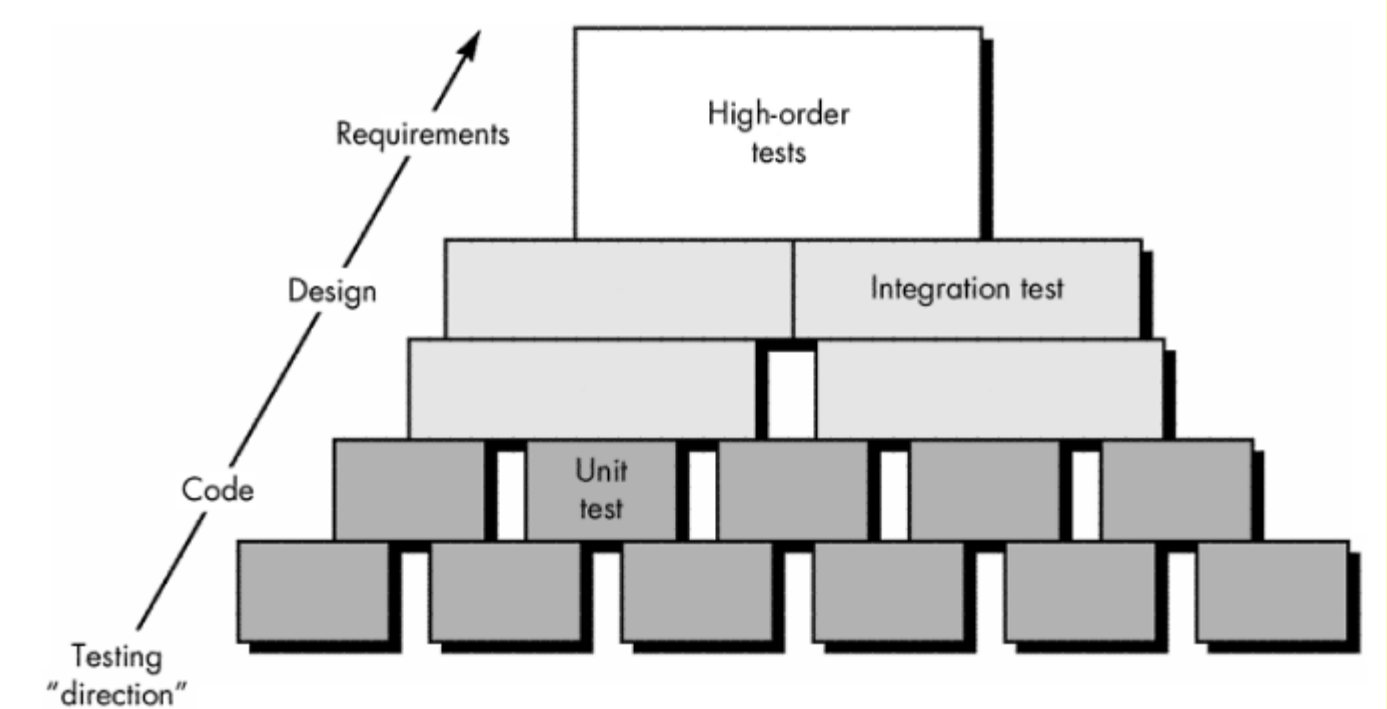
Testing strategy

- A strategy for software testing may be viewed as spiral.

A Software Testing Strategy

- Unit testing begins at the vortex of the spiral and concentrates on each unit (i.e., component) of the software as implemented in source code.
- Testing progresses by moving outward along the spiral to integration testing, where the focus is on design and the construction of the software architecture.
- Taking another turn outward on the spiral, we encounter validation testing, where requirements established as part of software requirements analysis are validated against the software that has been constructed.
- Finally, system testing, where the software and other system elements are tested as a whole.

A Software Testing Strategy



Software testing steps

- Unit test >> white-box techniques
- Integration test >> black-box + limited white-box
- Validation test >> black-box techniques

Strategic Issues

1. **Specify product requirements in a quantifiable manner long before testing commences.**
2. **State testing objectives explicitly.** The specific objectives of testing should be stated in measurable terms.
3. **Understand the users of the software and develop a profile for each user category.** Use-cases that describe the interaction scenario for each class of user can reduce overall testing effort by focusing testing on actual use of the product.
4. **Develop a testing plan that emphasizes "rapid cycle testing."**

Strategic Issues

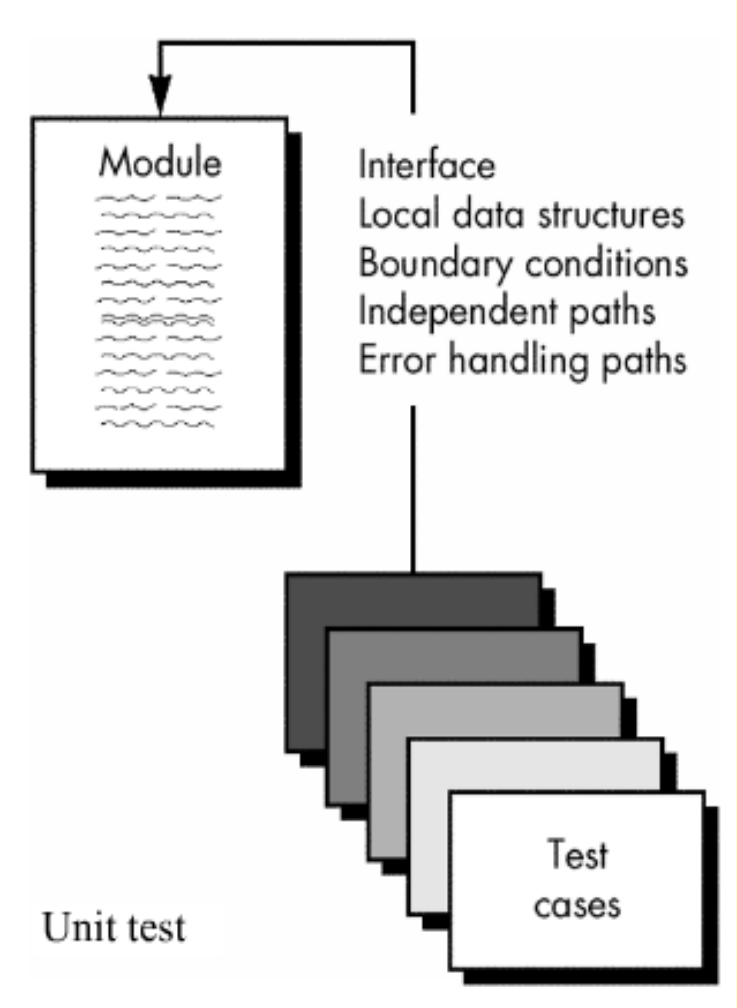
5. **Build "robust" software that is designed to test itself.** Software should be designed in a manner that uses anti-bugging techniques. That is, software should be capable of diagnosing certain classes of errors. In addition, the design should accommodate automated testing and regression testing.
6. **Use effective formal technical reviews as a filter prior to testing.**
7. **Conduct formal technical reviews to assess the test strategy and test cases themselves.**
8. **Develop a continuous improvement approach for the testing process.**

Unit Testing

- Unit testing focuses 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 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.

Unit Testing

- 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.
- All error handling paths are tested.

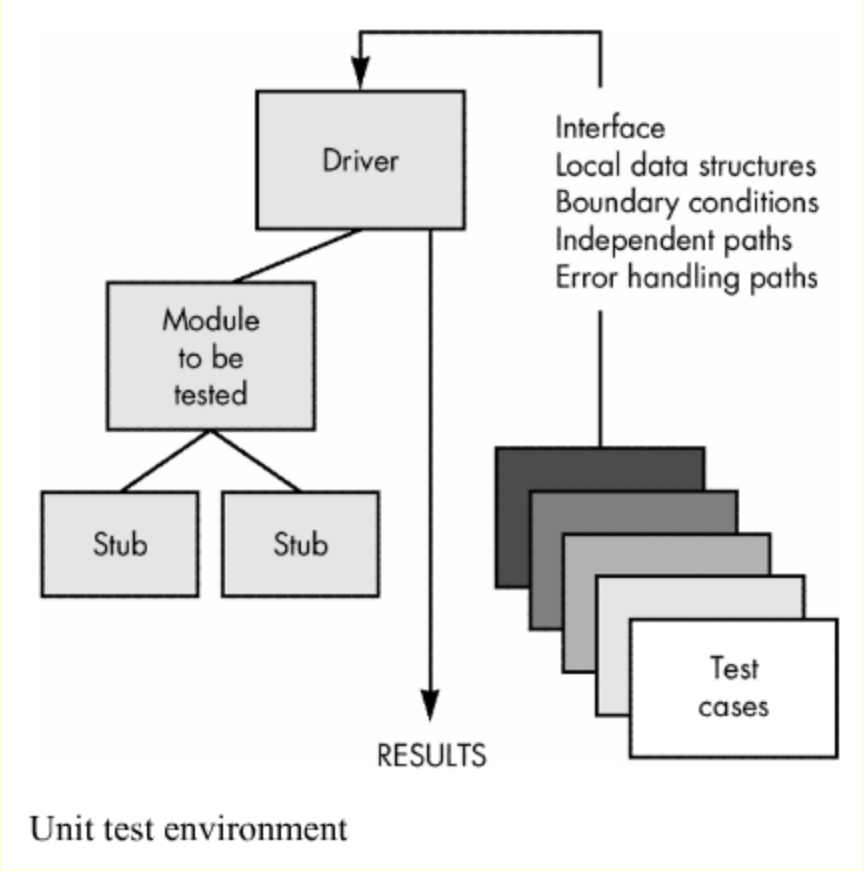


Unit Testing

- 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.
- 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.

Unit Testing

- Drivers and stubs represent overhead. That is, both are software that must be written.
- There are some situations in which you will not have the resources to do comprehensive unit testing. Select critical modules and those with high cyclomatic complexity and unit test only them.



Unit test environment

Integration Testing

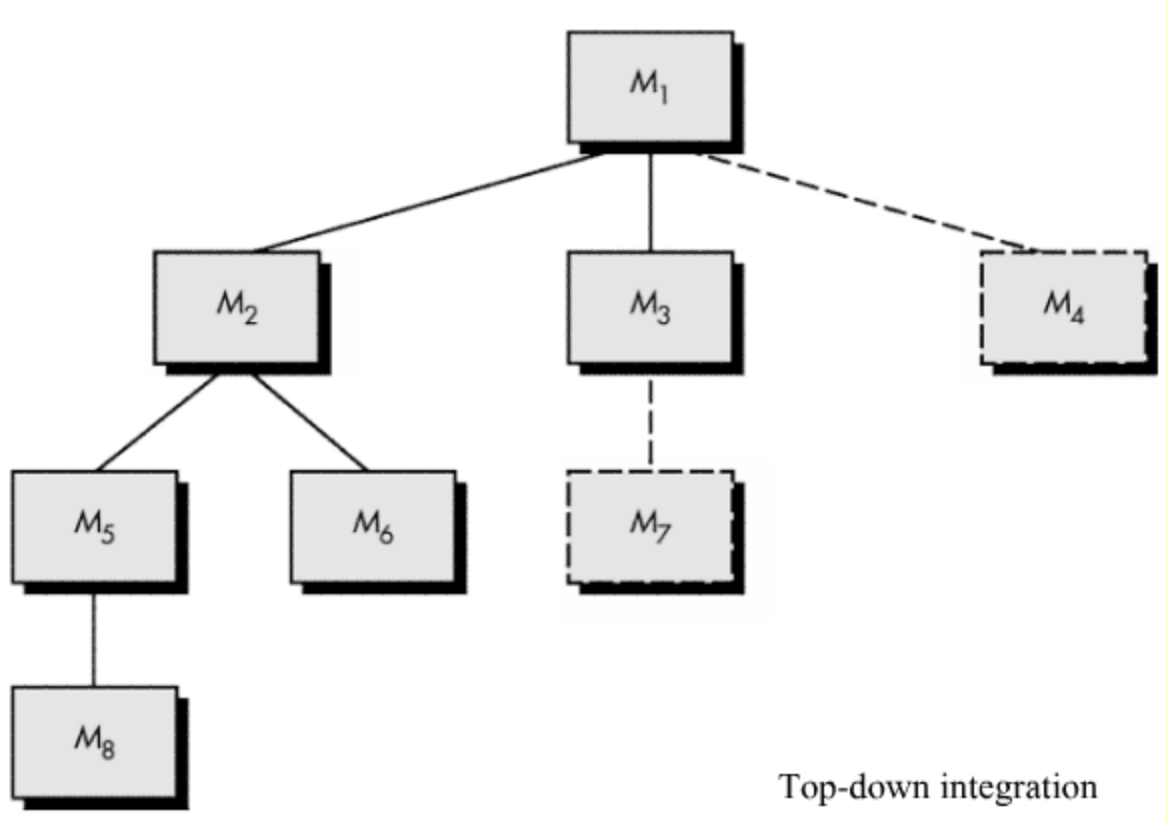
- Integration testing is a systematic technique for constructing the program structure while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit tested components and build a program structure that has been designed.
- Taking the big bang approach to integration is a lazy strategy that is doomed to failure. Integration testing should be conducted incrementally.

Integration Testing

- **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. 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.
- Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally.

Integration Testing

- Depth-first integration: M_1, M_2 , M_5 would be integrated first. Next, M_8 or M_6



- Breadth-first integration: M_2, M_3, and M_4 would be integrated first. The next control level, M_5, M_6, and so on, follows.

Integration Testing

- **Top-down integration testing – 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.

Integration Testing

- **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; therefore, no significant data can flow upward in the program structure.
- The tester is left with three choices: (1) delay many tests until stubs are replaced with actual modules, (2) develop stubs that perform limited functions that simulate the actual module, or (3) integrate the software from the bottom of the hierarchy upward.

Integration Testing

- **Bottom-up Integration** begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure).

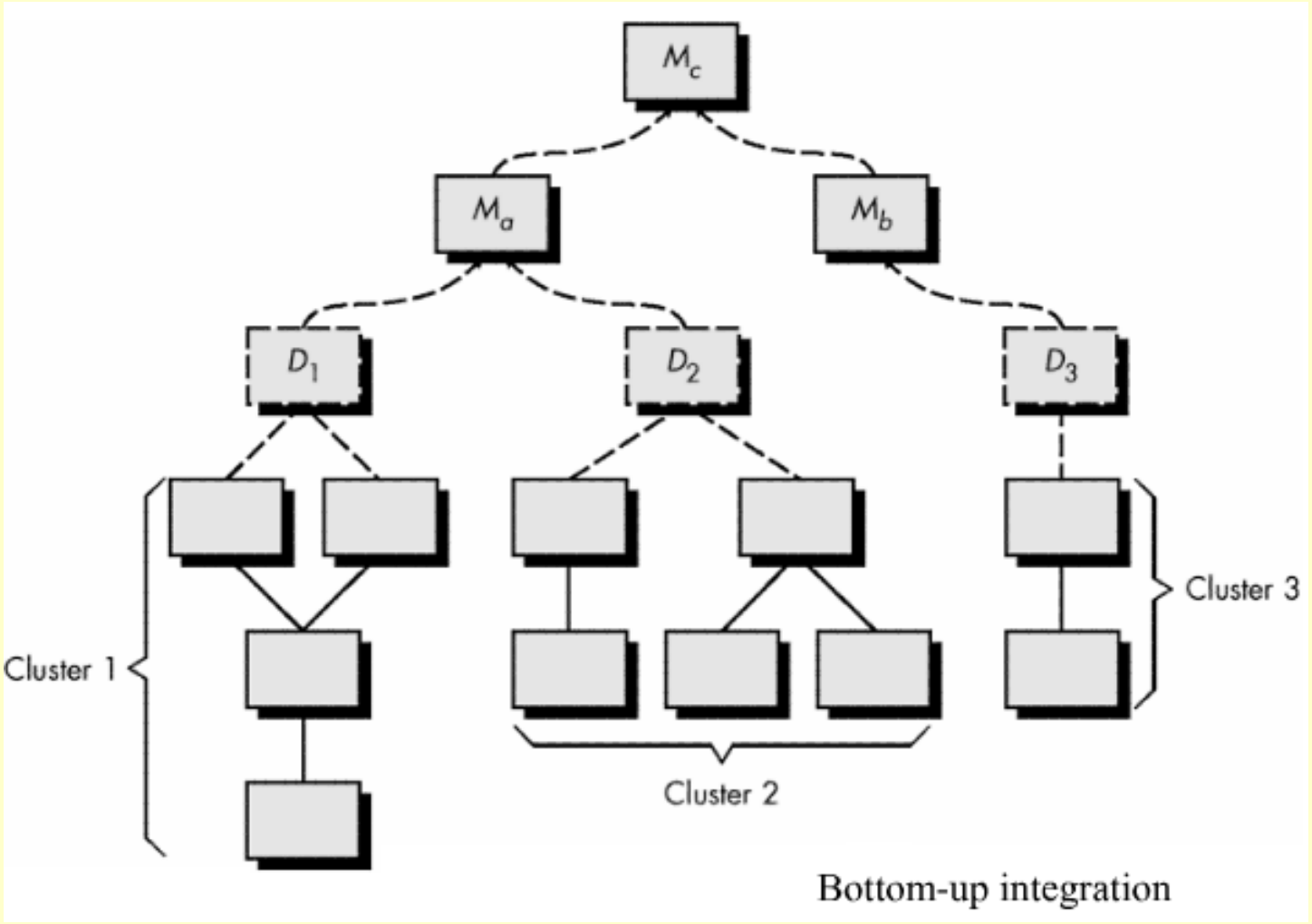
- Because components are integrated from the bottom up, processing required for components subordinate to a given level is always available and the need for stubs is eliminated.

Integration Testing

- **Bottom-up integration testing – steps:**

1. Low-level components are combined into clusters (sometimes called *builds*) 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 Testing



Bottom-up integration

Integration Testing

- **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.

Integration Testing

- **Smoke Testing** is an integration testing approach for time-critical projects, allowing the software team to assess its project on a frequent basis.

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.
3. The build is integrated with other builds and the entire product is smoke tested daily.

Integration Testing

- **Smoke testing – benefits:**

1. Integration risk is minimized.
2. The quality of the end-product is improved.
3. Error diagnosis and correction are simplified - errors uncovered during smoke testing are likely to be associated with "new software increments.
4. Progress is easier to assess.

Integration Testing

- An overall plan for integration of the software and a description of specific tests are documented in a Test Specification. This document 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.

Validation Testing

- Like all other testing steps, validation tries to uncover errors, but the focus is at the requirements level.
- 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.
- 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 or (2) a deviation from specification is uncovered and a **deficiency list** is created.

Validation Testing

- Configuration review or audit is used to ensure that all elements of the software configuration have been properly developed, cataloged, and documented to allow its support during its maintenance phase.
- When custom software is built for one customer, a series of acceptance tests are conducted to enable the customer to validate all requirements.
- If software is developed as a product to be used by many customers, it is impractical to perform formal acceptance tests with each one.

Validation Testing

- The alpha test is conducted at the developer's site by a customer.

The developers are present during testing.

The developer records the 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.

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

System 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.
- **Security testing:** attempts to verify that protection mechanisms built into a system will, in fact, protect it from improper penetration.
- **Stress testing:** executes a system in a manner that demands resources in abnormal quantity, frequency, or volume (e.g. special tests may be designed that generate ten interrupts per second, when one or two is the average rate).

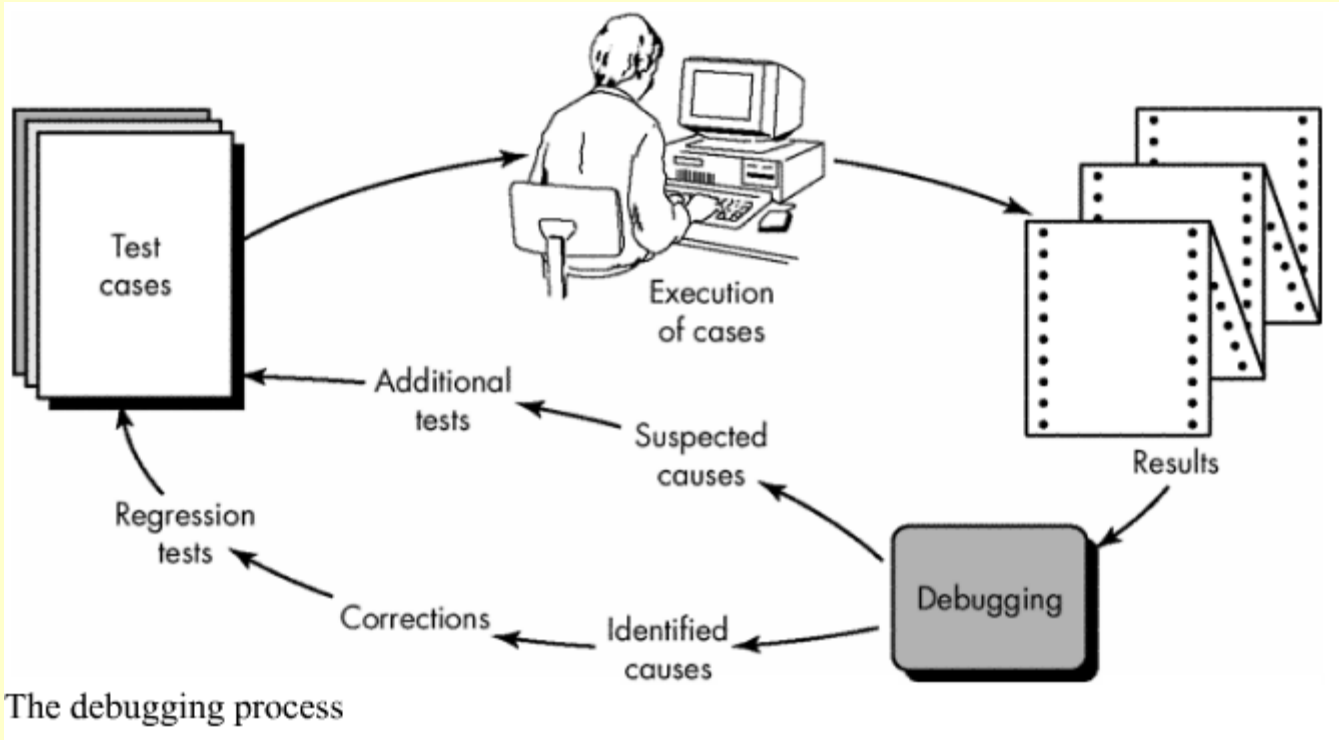
System Testing

- For real-time and embedded systems, software that provides required function but does not conform to performance requirements is unacceptable.
- **Performance testing** is designed to test the run-time performance of software within the context of an integrated system.
- Performance tests are often coupled with stress testing and usually require both hardware and software instrumentation.

The Art of Debugging

- 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.

Debugging is not testing but always occurs as a consequence of testing.



The debugging process

The Art of Debugging

- **Why is debugging so difficult?**

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 non-errors (e.g., round-off inaccuracies).
4. The symptom may be caused by human error that is not easily traced.

The Art of Debugging

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.

The Art of Debugging

- Three categories for debugging approaches may be proposed: (1) brute force, (2) backtracking, and (3) cause elimination.
- 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. We hope that somewhere in the morass of information that is produced we will find a clue that can lead us to the cause of an error.

The Art of Debugging

- **Backtracking** is a fairly common debugging approach that can be used 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.
- **Cause elimination**. 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. Alternatively, a list of all possible causes is developed and tests are conducted to eliminate each.

The Art of Debugging

- Simple questions that should be asked before making the "correction" that removes the cause of a bug:

1. **Is the cause of the bug reproduced in another part of the program?**
2. **What "next bug" might be introduced by the fix I'm about to make?** Before the correction is made, the source code (or, better, the design) should be evaluated to assess coupling of logic and data structures.
3. **What could we have done to prevent this bug in the first place?**

End of Testing chapter