

Software Testing Strategies

Slide Set - 14

**Organized & Presented By:
Software Engineering Team CSED
TIET, Patiala**

Software Engineering: A Practitioner's Approach, 6/e

Chapter 13

Software Testing Strategies

copyright © 1996, 2001, 2005

R.S. Pressman & Associates, Inc.

For University Use Only

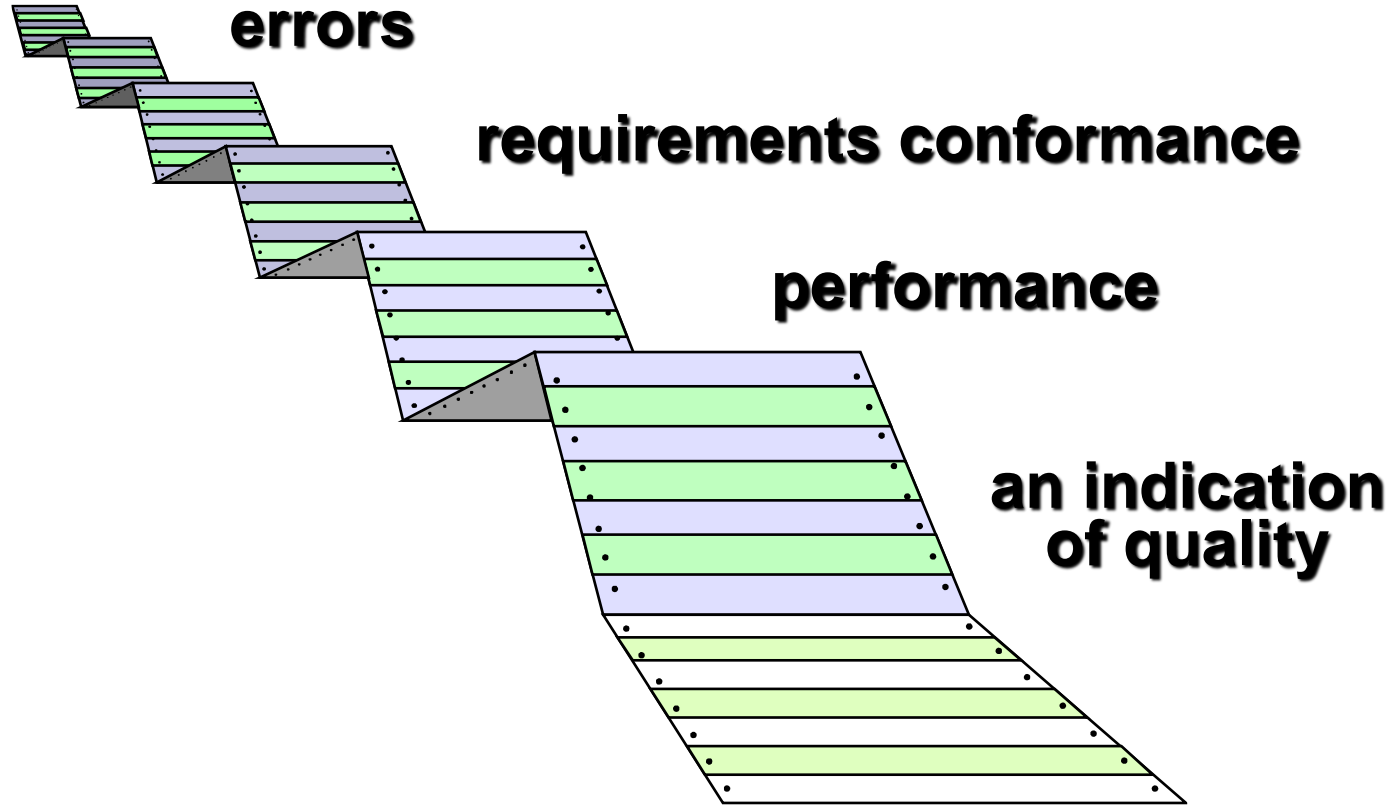
May be reproduced ONLY for student use at the university level
when used in conjunction with *Software Engineering: A Practitioner's Approach*.

Any other reproduction or use is expressly prohibited.

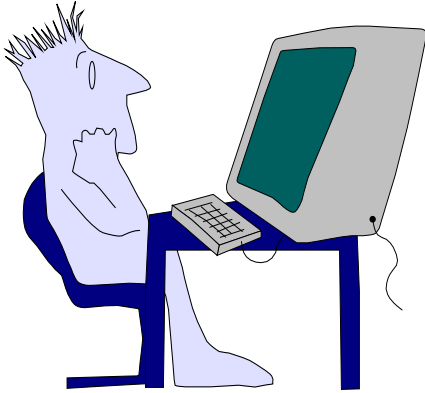
Software Testing

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

What Testing Shows

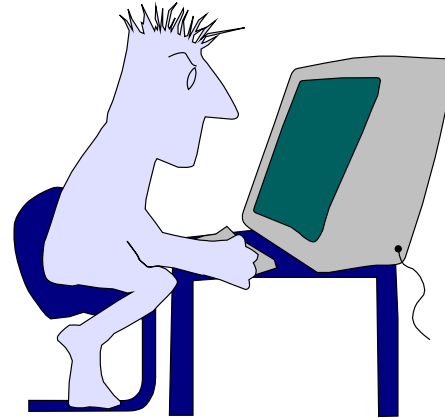


Who Tests the Software?



developer

**Understands the system
but, will test "gently"
and, is driven by "delivery"**



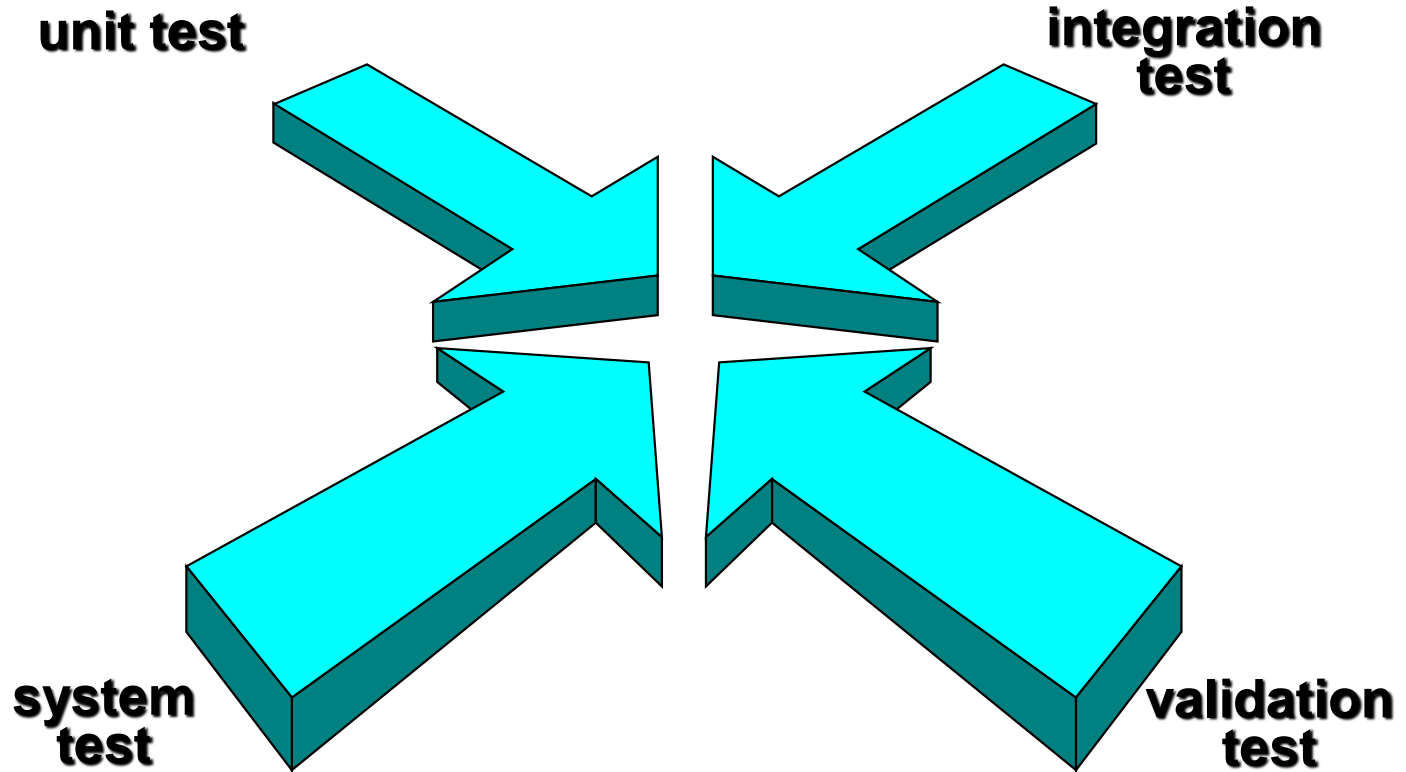
independent tester

**Must learn about the system,
but, will attempt to break it
and, is driven by quality**

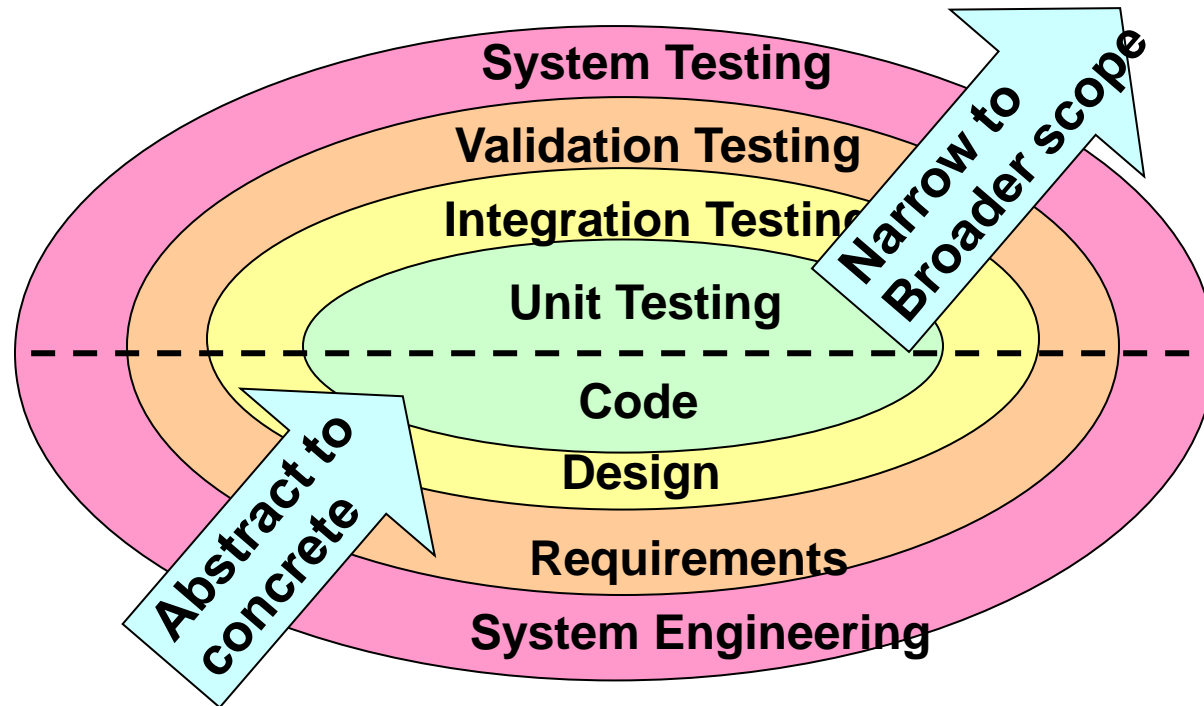
Verification and Validation

- Software testing is part of a broader group of activities called verification and validation that are involved in software quality assurance
- **Verification** (Are the algorithms coded correctly?)
 - The set of activities that ensure that software correctly implements a specific function or algorithm
- **Validation** (Does it meet user requirements?)
 - The set of activities that ensure that the software that has been built is traceable to customer requirements

Testing Strategy



A Strategy for Testing Conventional Software



Levels of Testing for Conventional Software

- Unit testing
 - Concentrates on each component/function of the software as implemented in the source code
- Integration testing
 - Focuses on the design and construction of the software architecture
- Validation testing
 - Requirements are validated against the constructed software
- System testing
 - The software and other system elements are tested as a whole

Testing Strategy applied to Conventional Software

- Unit testing
 - Exercises specific paths in a component's control structure to ensure complete coverage and maximum error detection
 - Components are then assembled and integrated
- Integration testing
 - Focuses on inputs and outputs, and how well the components fit together and work together
- Validation testing
 - Provides final assurance that the software meets all functional, behavioral, and performance requirements
- System testing
 - Verifies that all system elements (software, hardware, people, databases) mesh properly and that overall system function and performance is achieved

Testing Strategy

- We begin by ‘testing-in-the-small’ and move toward ‘testing-in-the-large’
- For conventional software
 - The module (component) is our initial focus
 - Integration of modules follows
- For OO software
 - our focus when “testing in the small” changes from an individual module (the conventional view) to an OO class that encompasses attributes and operations and implies communication and collaboration

Strategic Issues

- 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.

When is Testing Complete?

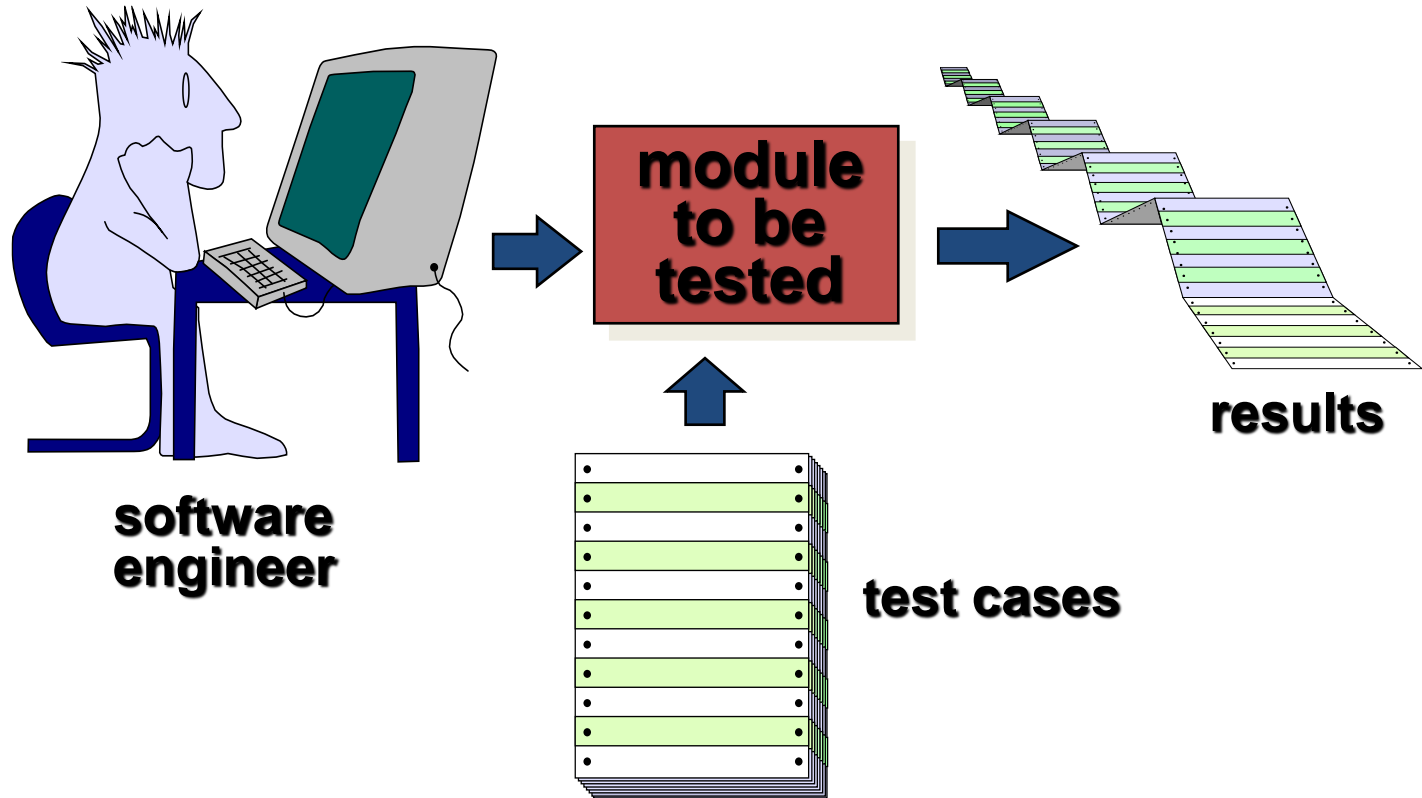
- There is no definitive answer to this question
- Every time a user executes the software, the program is being tested
- Sadly, testing usually stops when a project is running out of time, money, or both
- One approach is to divide the test results into various severity levels
 - Then consider testing to be complete when certain levels of errors no longer occur or have been repaired or eliminated

Ensuring a Successful Software Test Strategy

- Specify product requirements in a **quantifiable** manner long before testing commences
- State **testing objectives** explicitly in measurable terms
- Understand the user of the software (through use cases) and develop a **profile for each user category**
- Develop a **testing plan** that emphasizes rapid cycle testing to get quick feedback to control quality levels and adjust the test strategy
- Build robust software that is **designed to test itself** and can diagnose certain kinds of errors
- Use effective **formal technical reviews** as a filter prior to testing to reduce the amount of testing required
- Conduct formal technical reviews to **assess the test strategy** and test cases themselves
- Develop a **continuous improvement** approach for the testing process through the gathering of metrics

Test Strategies for Conventional Software

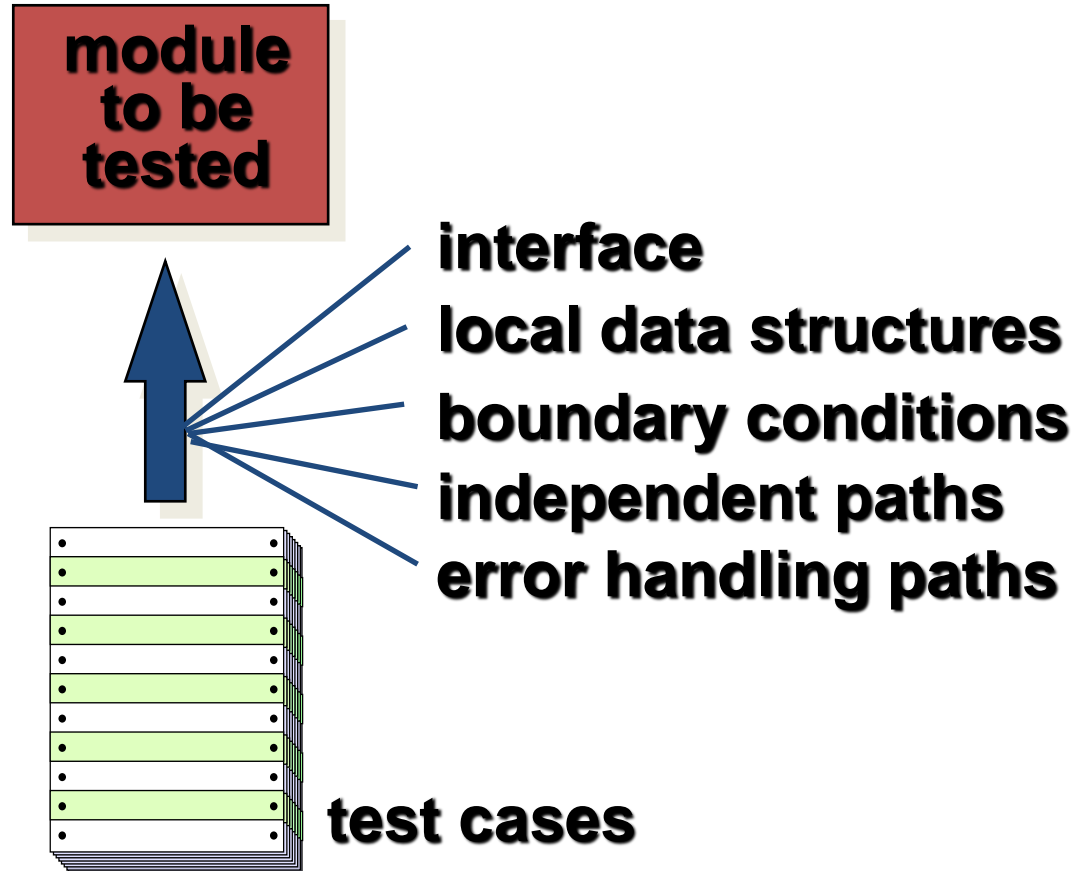
Unit Testing



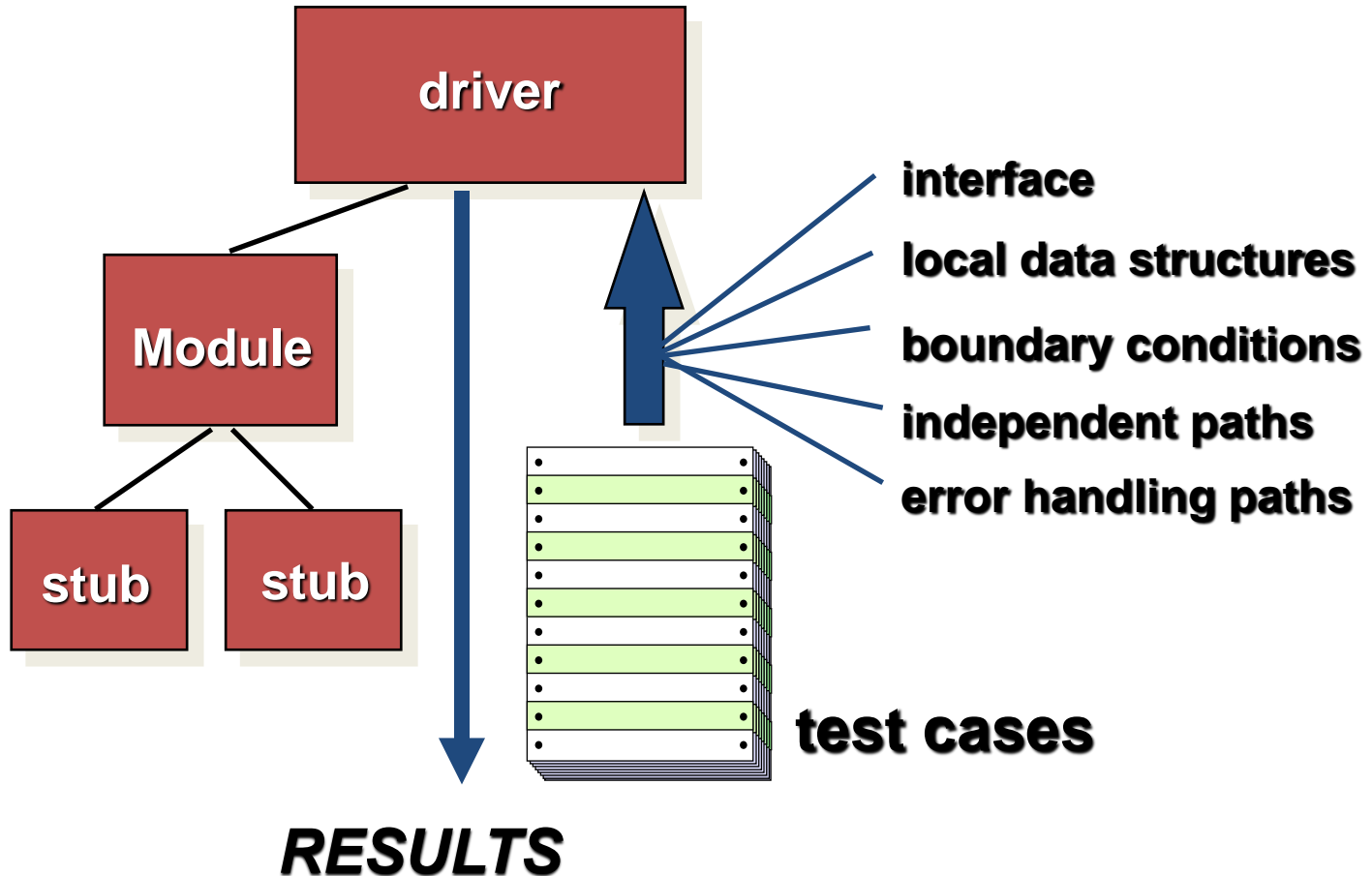
Unit Testing

- Focuses testing on the function or software module
- Concentrates on the internal processing logic and data structures
- Is simplified when a module is designed with high cohesion
 - Reduces the number of test cases
 - Allows errors to be more easily predicted and uncovered
- Concentrates on critical modules and those with **high cyclomatic complexity** when testing resources are limited

Unit Testing



Unit Test Environment



Targets for Unit Test Cases

- **Module interface**
 - Ensure that information flows properly into and out of the module
- **Local data structures**
 - Ensure that data stored temporarily maintains its integrity during all steps in an algorithm execution
- **Boundary conditions**
 - Ensure that the module operates properly at boundary values established to limit or restrict processing
- **Independent paths (basis paths)**
 - Paths are exercised to ensure that all statements in a module have been executed at least once
- **Error handling paths**
 - Ensure that the algorithms respond correctly to specific error conditions

Common Computational Errors in Execution Paths

- Misunderstood or incorrect arithmetic precedence
- Mixed mode operations (e.g., int, float, char)
- Incorrect initialization of values
- Precision inaccuracy and round-off errors
- Incorrect symbolic representation of an expression (int vs. float)

Other Errors to Uncover

- Comparison of different data types
- Incorrect logical operators or precedence
- Expectation of equality when precision error makes equality unlikely (using `==` with float types)
- Incorrect comparison of variables
- Improper or nonexistent loop termination
- Failure to exit when divergent iteration is encountered
- Improperly modified loop variables
- Boundary value violations

Problems to uncover in Error Handling

- Error description is unintelligible or ambiguous
- Error noted does not correspond to error encountered
- Error condition causes operating 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

Drivers and Stubs for Unit Testing

- **Driver**
 - A simple main program that accepts test case data, passes such data to the component being tested, and prints the returned results
- **Stubs**
 - Serve to replace modules that are subordinate to (called by) the component to be tested
 - It uses the module's exact interface, may do minimal data manipulation, provides verification of entry, and returns control to the module undergoing testing
- **Drivers and stubs both represent overhead**
 - Both must be written but don't constitute part of the installed software product

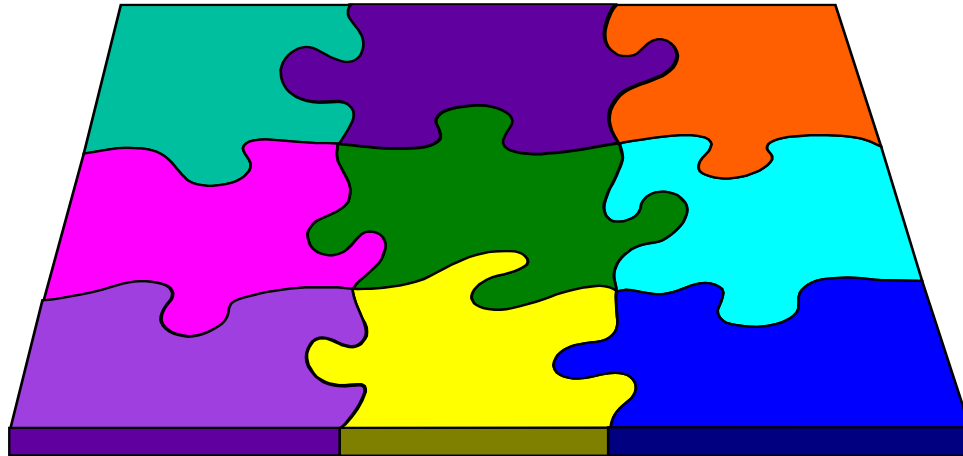
Integration Testing

- Defined as a systematic technique for constructing the software architecture
 - At the same time integration is occurring, conduct tests to uncover errors associated with interfaces
- Objective is to take unit tested modules and build a program structure based on the prescribed design
- Two Approaches
 - **Non-incremental Integration Testing**
 - **Incremental Integration Testing**

Integration Testing Strategies

Options:

- the “big bang” approach
- an incremental construction strategy



Non-incremental Integration Testing

- Commonly called the “Big Bang” approach
- All components are combined in advance
- The entire program is tested as a whole
- Chaos results
- Many seemingly-unrelated errors are encountered
- Correction is difficult because isolation of causes is complicated
- Once a set of errors are corrected, more errors occur, and testing appears to enter an endless loop

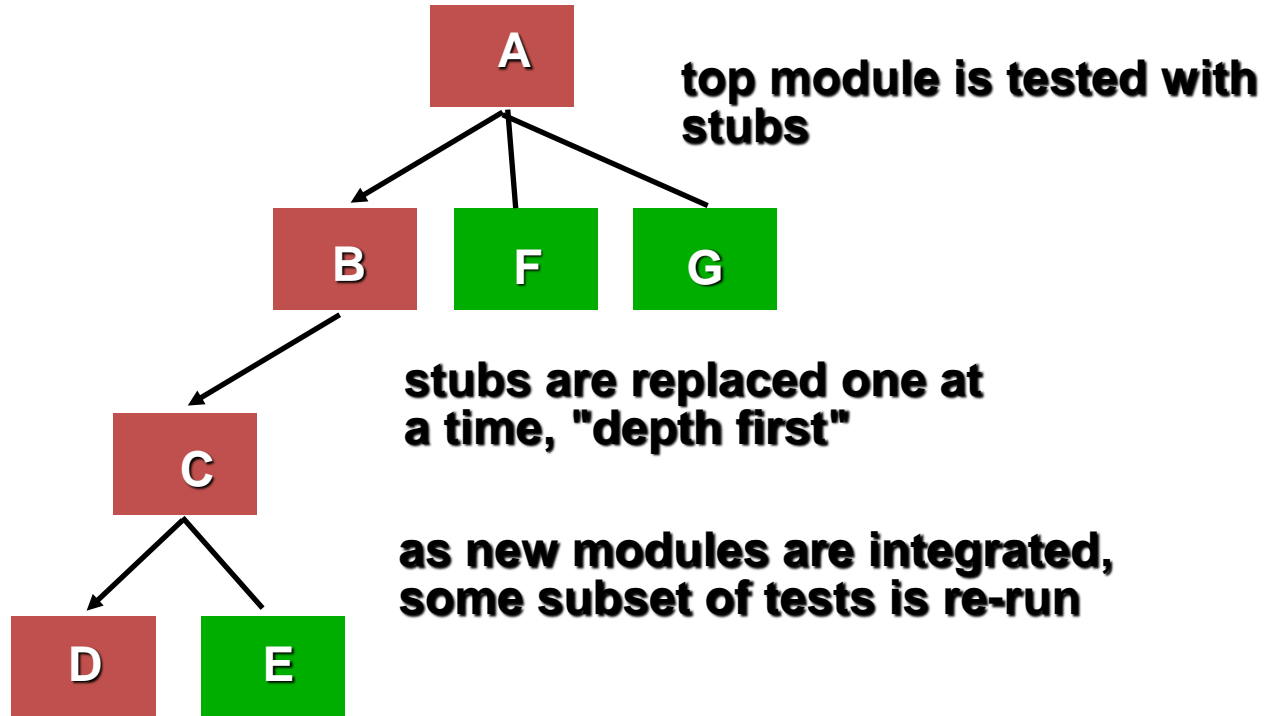
Incremental Integration Testing

- Three kinds
 - Top-down integration
 - Bottom-up integration
 - Sandwich integration
- The program is constructed and tested in small increments
- Errors are easier to isolate and correct
- Interfaces are more likely to be tested completely
- A systematic test approach is applied

Top-down Integration

- Modules are integrated by moving downward through the control hierarchy, beginning with the main module
- Subordinate modules are incorporated in either a depth-first or breadth-first fashion
 - DF: All modules on a major control path are integrated
 - BF: All modules directly subordinate at each level are integrated
- 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

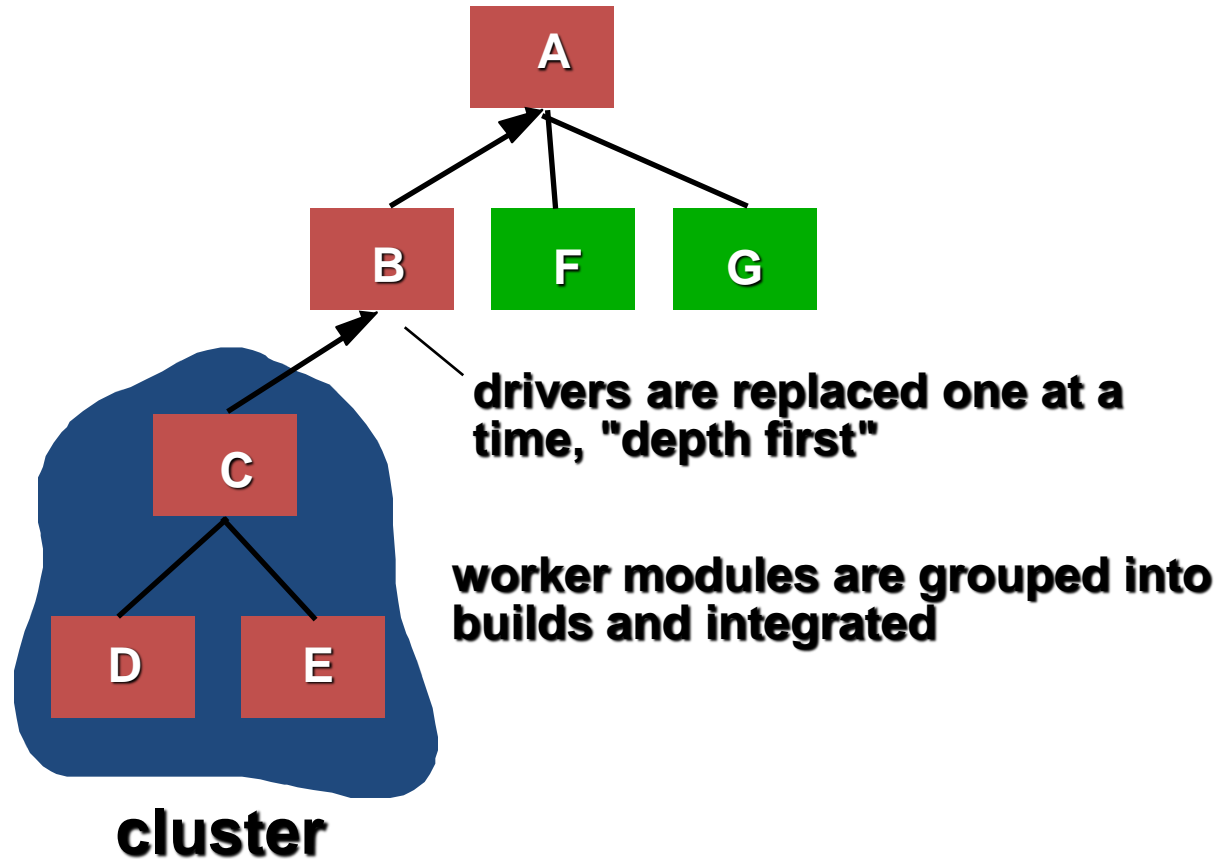
Top Down Integration



Bottom-up Integration

- Integration and testing starts with the most atomic modules in the control hierarchy
- Advantages
 - This approach verifies low-level data processing early in the testing process
 - Need for stubs is eliminated
- Disadvantages
 - Driver modules need to be built to test the lower-level modules; this code is later discarded or expanded into a full-featured version
 - Drivers inherently do not contain the complete algorithms that will eventually use the services of the lower-level modules; consequently, testing may be incomplete or more testing may be needed later when the upper level modules are available

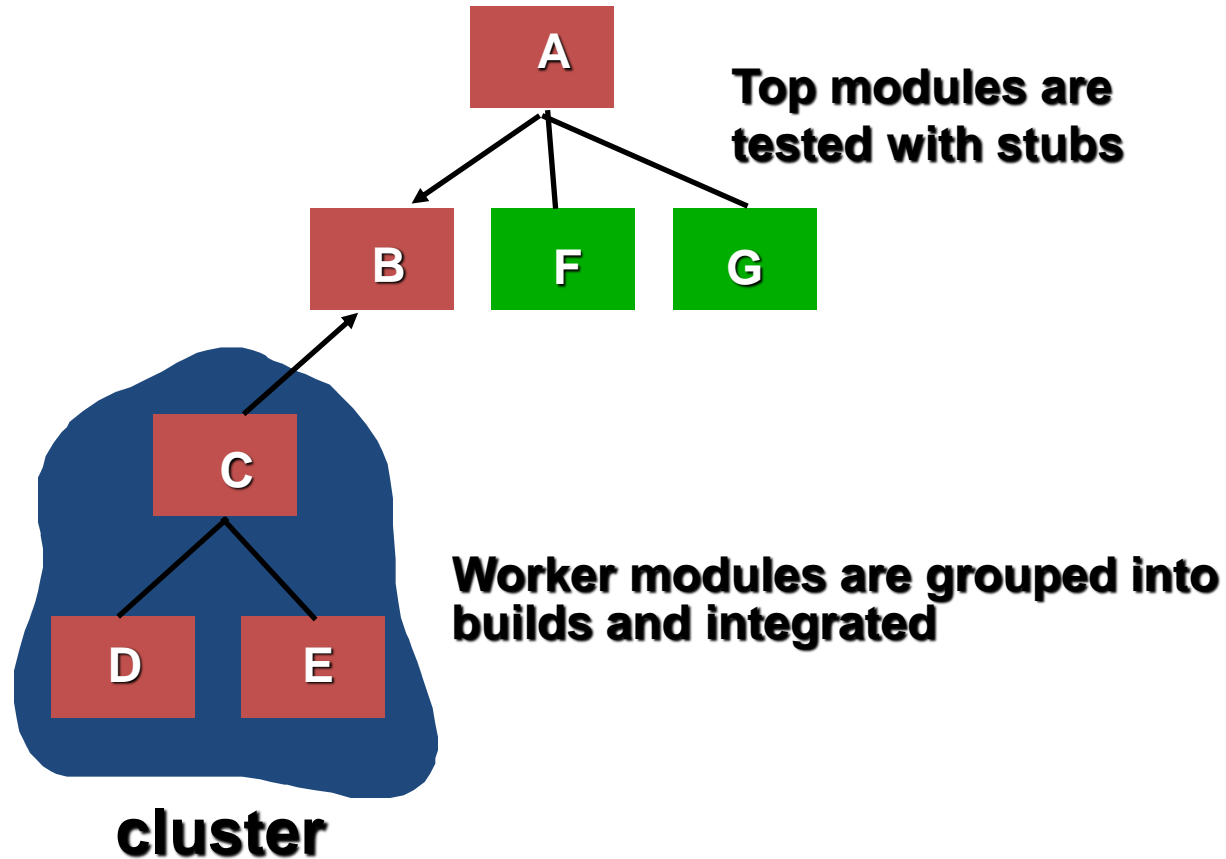
Bottom-Up Integration



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

Sandwich Testing



Object-Oriented Testing

- begins by evaluating the correctness and consistency of the OOA and OOD models
- testing strategy changes
 - the concept of the 'unit' broadens due to encapsulation
 - integration focuses on classes and their execution across a 'thread' or in the context of a usage scenario
 - 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

Regression Testing

- Each new addition or change to baselined software may cause problems with functions that previously worked flawlessly
- Regression testing re-executes a small subset of tests that have already been conducted
 - Ensures that changes have not propagated unintended side effects
 - Helps to ensure that changes do not introduce unintended behavior or additional errors
 - May be done manually or through the use of automated capture/playback tools
- Regression test suite contains three different 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 actual software components that have been changed

Smoke Testing

- Taken from the world of hardware
 - Power is applied and a technician checks for sparks, smoke, or other dramatic signs of fundamental failure
- Designed as a pacing mechanism for time-critical projects
 - Allows the software team to assess its project on a frequent basis
- Includes the following activities
 - The software is compiled and linked into a build
 - A series of breadth tests is designed to expose errors that will keep the build from properly performing its function
 - The goal is 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
 - Daily testing gives managers and practitioners a realistic assessment of the progress of the integration testing
 - After a smoke test is completed, detailed test scripts are executed

Smoke Testing Steps

- A common approach for creating “daily builds” for product software
- Smoke testing steps:
 - 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 (in its current form) is smoke tested daily.
 - The integration approach may be top down or bottom up.

Benefits of Smoke Testing

- Integration risk is minimized
 - Daily testing uncovers incompatibilities and show-stoppers early in the testing process, thereby reducing schedule impact
- The quality of the end-product is improved
 - Smoke testing is likely to uncover both functional errors and architectural and component-level design errors
- Error diagnosis and correction are simplified
 - Smoke testing will probably uncover errors in the newest components that were integrated
- Progress is easier to assess
 - As integration testing progresses, more software has been integrated and more has been demonstrated to work
 - Managers get a good indication that progress is being made

High Order Testing

- **Validation testing**
 - Focus is on software requirements
- **System testing**
 - Focus is on system integration
- **Alpha/Beta testing**
 - Focus is on customer usage
- **Recovery testing**
 - forces the software to fail in a variety of ways and verifies that recovery is properly performed
- **Security testing**
 - verifies 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
- **Performance Testing**
 - test the run-time performance of software within the context of an integrated system

Validation Testing

Background

Validation testing follows integration testing

- The distinction between conventional and object-oriented software disappears
- Focuses on user-visible actions and user-recognizable output from the system
- Demonstrates conformity with requirements
- Designed to ensure that
 - All functional requirements are satisfied
 - All behavioral characteristics are achieved
 - All performance requirements are attained
 - Documentation is correct
 - Usability and other requirements are met (e.g., transportability, compatibility, error recovery, maintainability)

After each validation test

- ☐ The function or performance characteristic conforms to specification and is accepted
- ☐ A deviation from specification is uncovered and a deficiency list is created

A configuration review or audit ensures that all elements of the software configuration have been properly developed, cataloged, and have the necessary detail for entering the support phase of the software life cycle

Alpha and Beta Testing

- **Alpha testing**
 - Conducted at the developer's site by end users
 - Software is used in a natural setting with developers watching intently
 - Testing is conducted in a controlled environment
- **Beta testing**
 - Conducted at end-user sites
 - Developer is generally not present
 - It serves as a live application of the software in an environment that cannot be controlled by the developer
 - The end-user records all problems that are encountered and reports these to the developers at regular intervals
- After beta testing is complete, software engineers make software modifications and prepare for release of the software product to the entire customer base

Different Types of System Testing

- **Recovery testing**
 - Tests for recovery from system faults
 - Forces the software to fail in a variety of ways and verifies that recovery is properly performed
 - Tests re-initialization, check pointing mechanisms, data recovery, and restart for correctness
- **Security testing**
 - Verifies that protection mechanisms built into a system will, in fact, protect it from improper access
- **Stress testing**
 - Executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- **Performance testing**
 - Tests the run-time performance of software within the context of an integrated system
 - Often coupled with stress testing and usually requires both hardware and software instrumentation
 - Can uncover situations that lead to degradation and possible system failure

The Art of Debugging

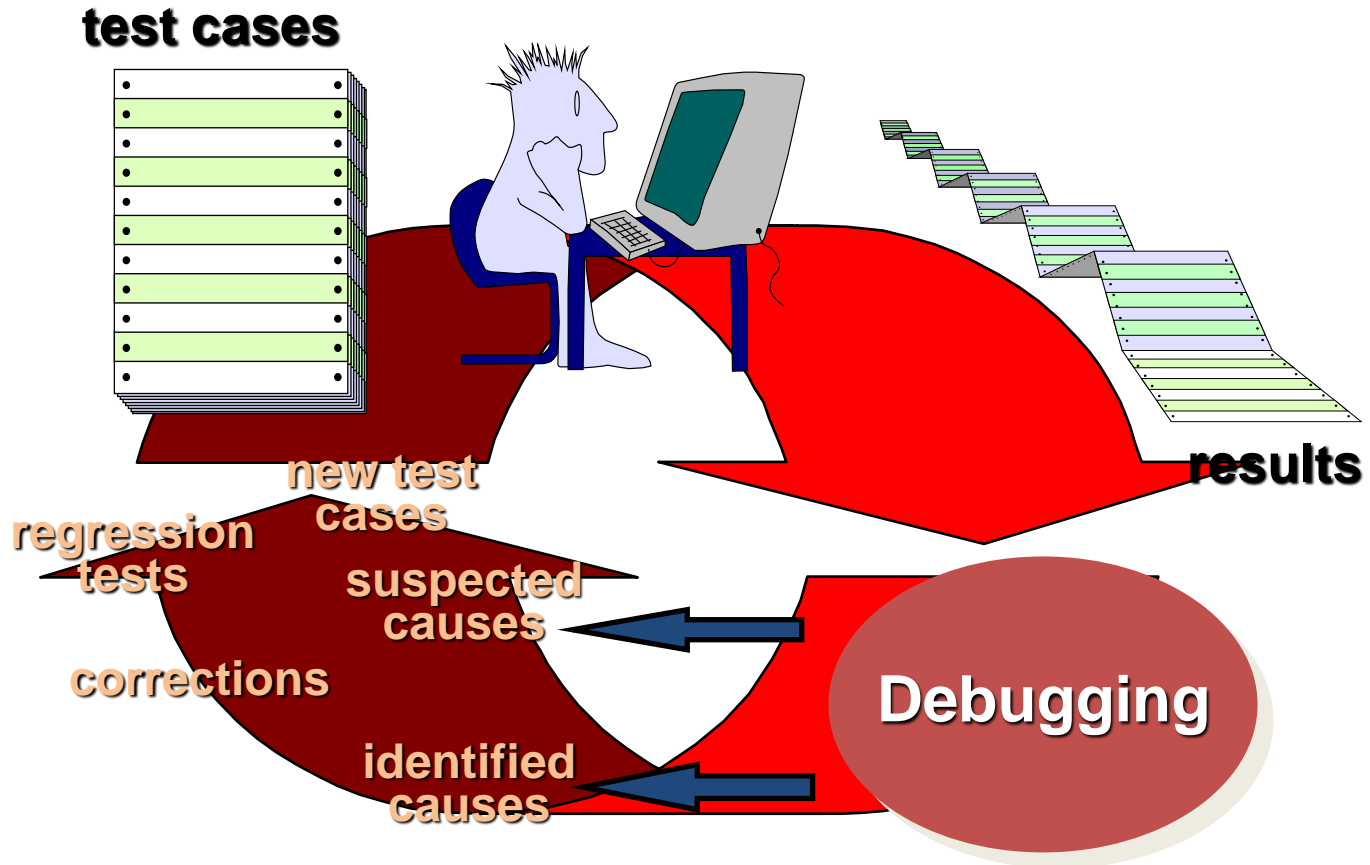
Debugging: A Diagnostic Process



Debugging Process

- Debugging occurs as a consequence of successful testing
- It is still very much an art rather than a science
- Good debugging ability may be an innate human trait
- Large variances in debugging ability exist
- The debugging process begins with the execution of a test case
- Results are assessed and the difference between expected and actual performance is encountered
- This difference is a symptom of an underlying cause that lies hidden
- The debugging process attempts to match symptom with cause, thereby leading to error correction

The Debugging Process



Why is Debugging so Difficult?

- The symptom and the cause may be geographically remote
- The symptom may disappear (temporarily) when another error is corrected
- The symptom may actually be caused by nonerrors (e.g., round-off accuracies)
- The symptom may be caused by human error that is not easily traced

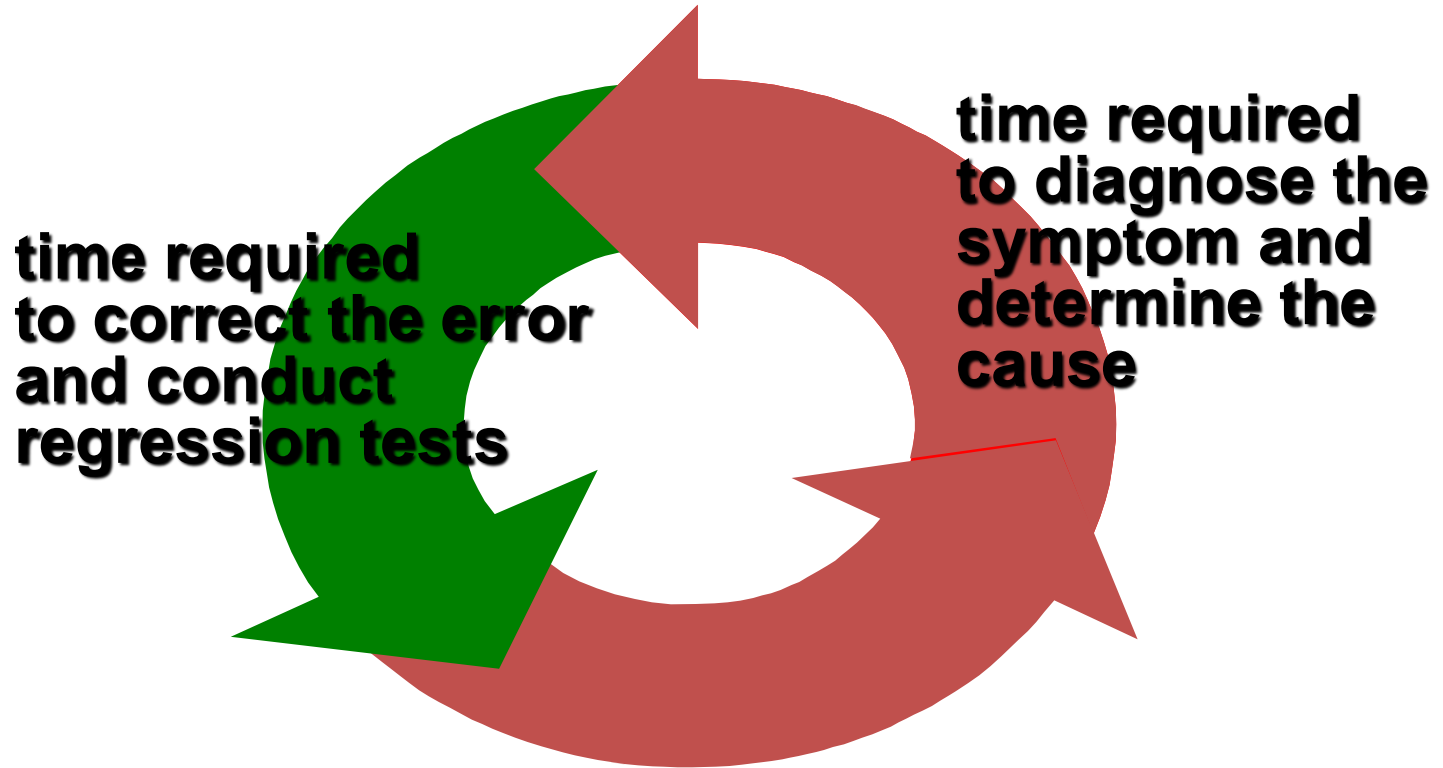
(continued on next slide)

Why is Debugging so Difficult?

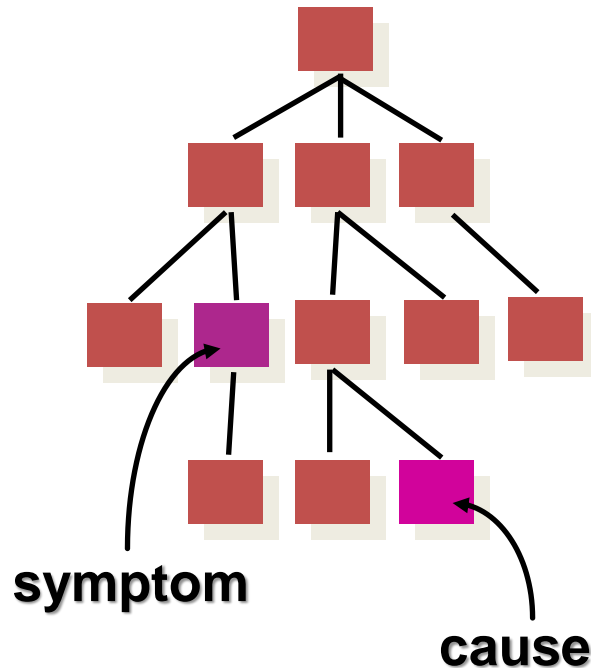
(continued)

- The symptom may be a result of timing problems, rather than processing problems
- It may be difficult to accurately reproduce input conditions, such as asynchronous real-time information
- The symptom may be intermittent such as in embedded systems involving both hardware and software
- The symptom may be due to causes that are distributed across a number of tasks running on different processes

Debugging Effort

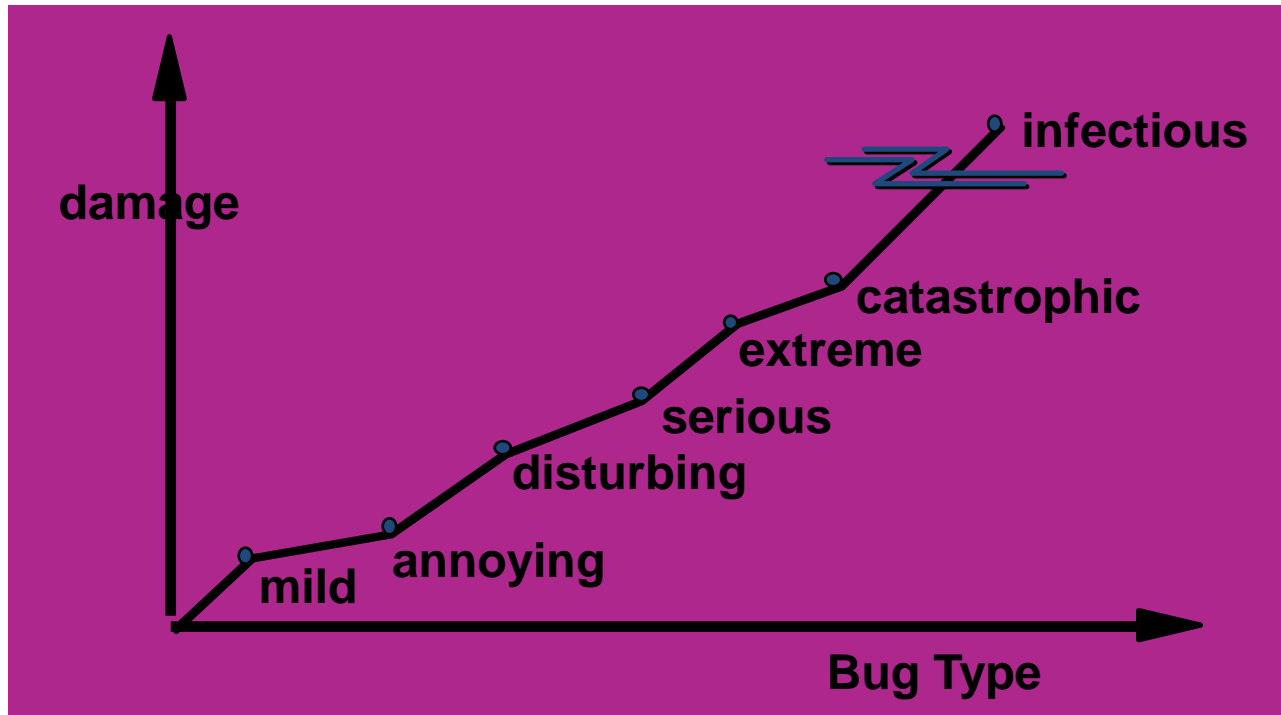


Symptoms & Causes



- ❑ 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

Consequences of Bugs



Bug Categories: function-related bugs, system-related bugs, data bugs, coding bugs, design bugs, documentation bugs, standards violations, etc.

Debugging Techniques

- ❑ **brute force / testing**
- ❑ **backtracking**
- ❑ **induction**
- ❑ **deduction**

Debugging Strategies

- Objective of debugging is to find and correct the cause of a software error
- Bugs are found by a combination of systematic evaluation, intuition, and luck
- Debugging methods and tools are not a substitute for careful evaluation based on a complete design model and clear source code
- There are three main debugging strategies
 - Brute force
 - Backtracking
 - Cause elimination

Strategy #1: Brute Force

- Most commonly used and least efficient method
- Used when all else fails
- Involves the use of memory dumps, run-time traces, and output statements
- Leads many times to wasted effort and time

Strategy #2: Backtracking

- Can be used successfully in small programs
- The method starts at the location where a symptom has been uncovered
- The source code is then traced backward (manually) until the location of the cause is found
- In large programs, the number of potential backward paths may become unmanageably large

Strategy #3: Cause Elimination

- Involves the use of induction or deduction and introduces the concept of binary partitioning
 - Induction (specific to general): Prove that a specific starting value is true; then prove the general case is true
 - Deduction (general to specific): Show that a specific conclusion follows from a set of general premises
- 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 cause
- If initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug

Three Questions to ask Before Correcting the Error

- Is the cause of the bug reproduced in another part of the program?
 - Similar errors may be occurring in other parts of the program
- What next bug might be introduced by the fix that I'm about to make?
 - The source code (and even the design) should be studied to assess the coupling of logic and data structures related to the fix
- What could we have done to prevent this bug in the first place?
 - This is the first step toward software quality assurance
 - By correcting the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs



Debugging: Final Thoughts

1. Don't run off half-cocked, think about the symptom you're seeing.
2. Use tools (e.g., dynamic debugger) to gain more insight.
3. If at an impasse, get help from someone else.
4. Be absolutely sure to conduct regression tests when you do "fix" the bug.