

Software Engineering

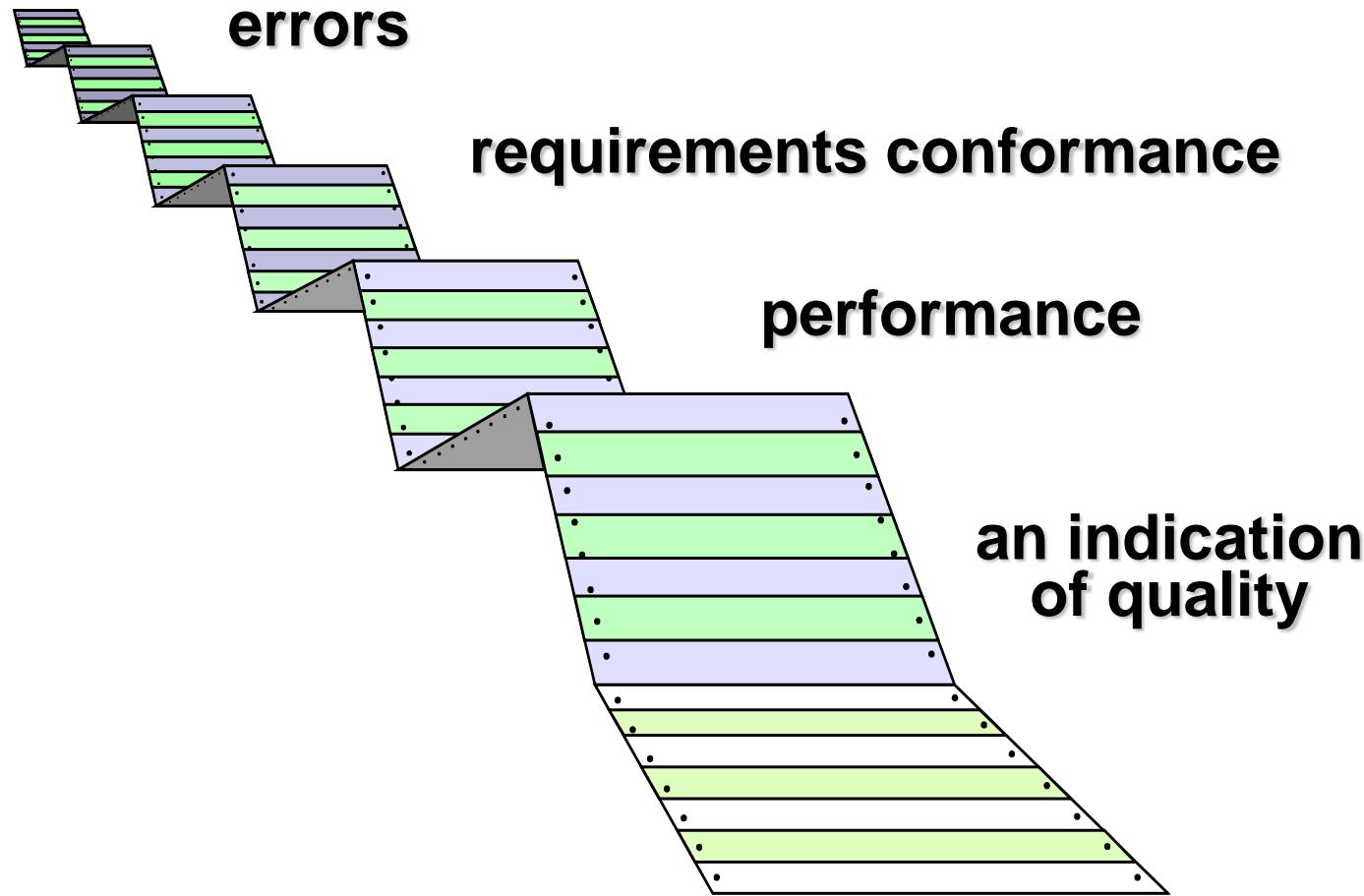
Chapter 13 Software Testing Strategies



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

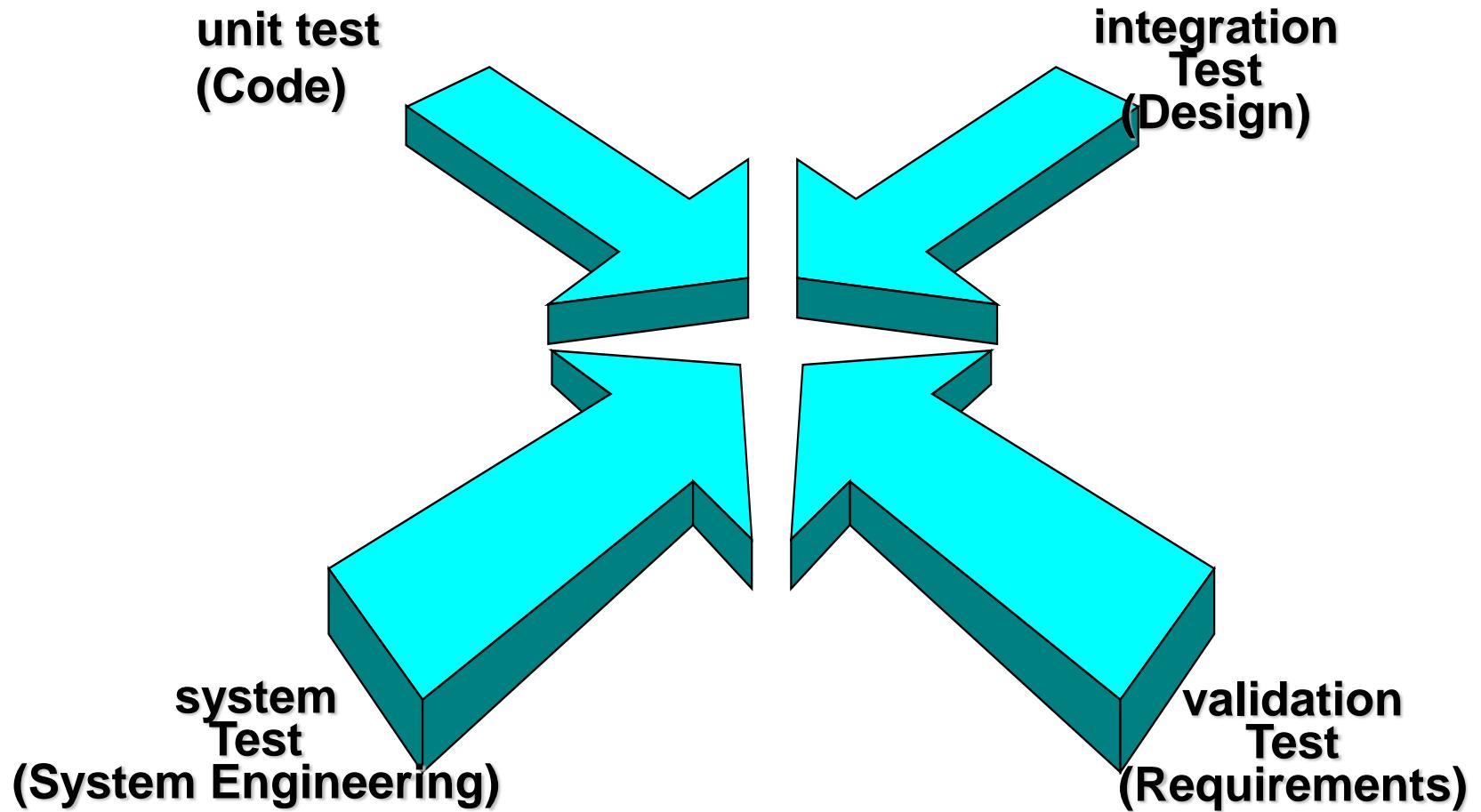
**Understands the system
but, will test "gently"
and, is driven by "delivery"**



independent tester

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

Testing Strategy



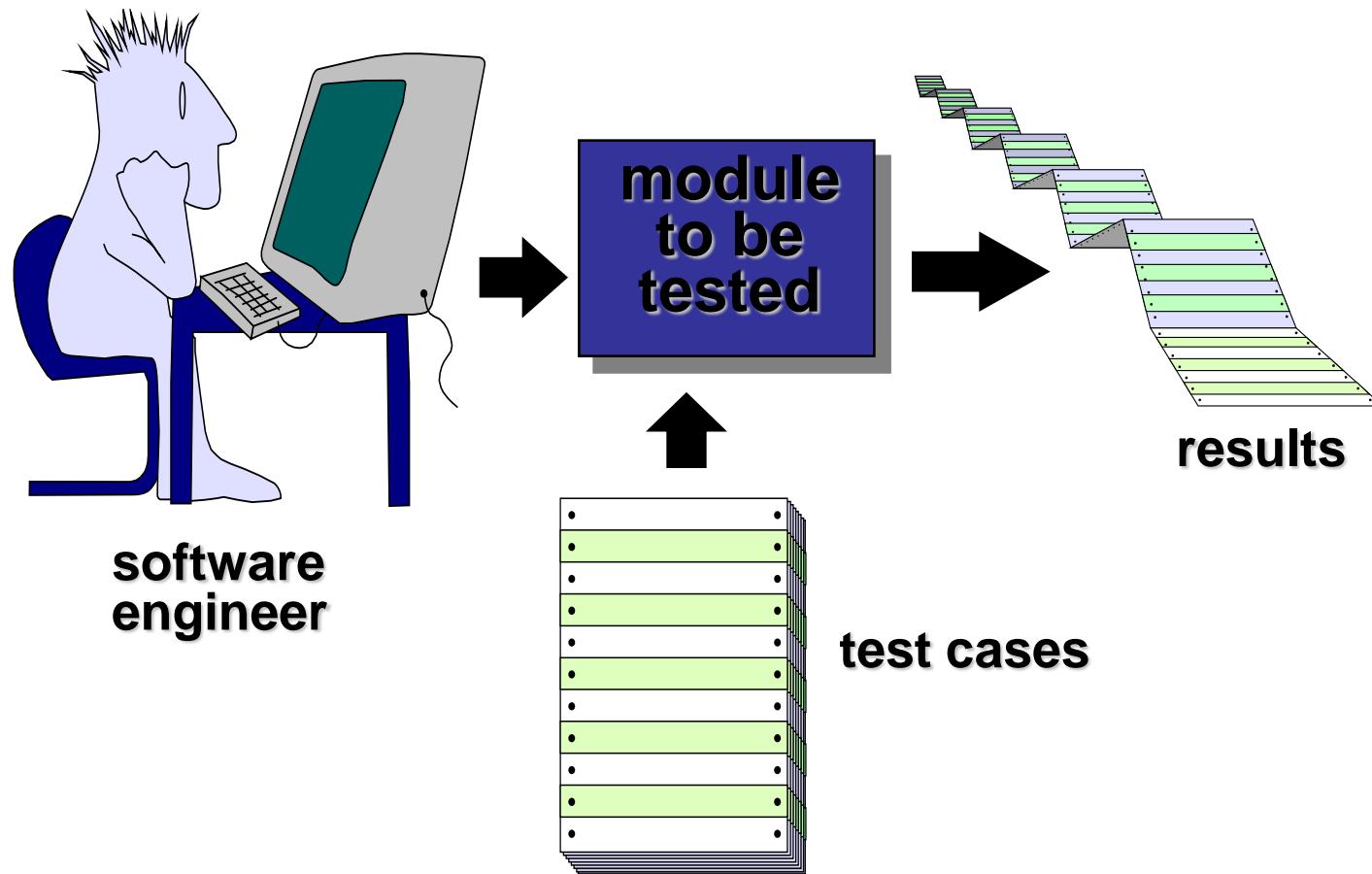
Testing Strategy

- 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

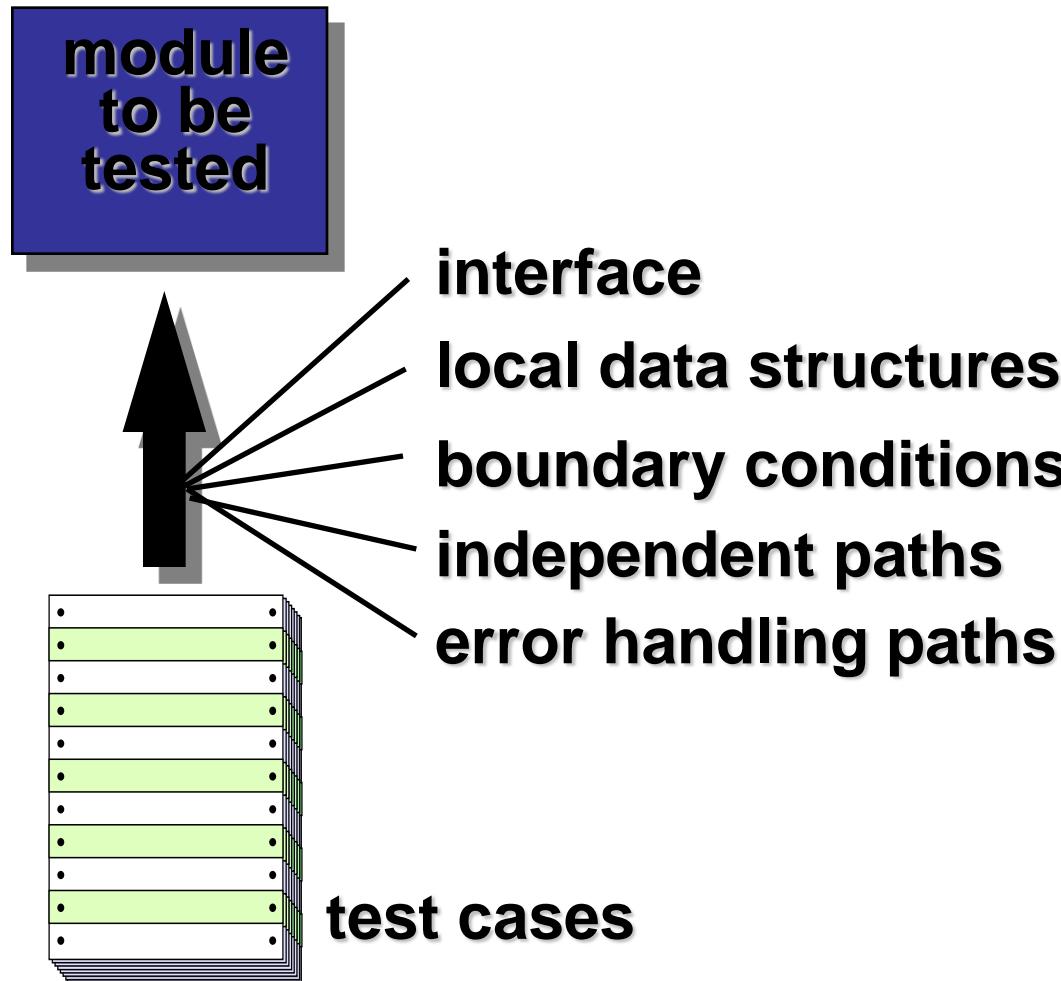
Strategic Issues

- Specify product requirements in a quantifiable manner long before testing commence.
- 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.

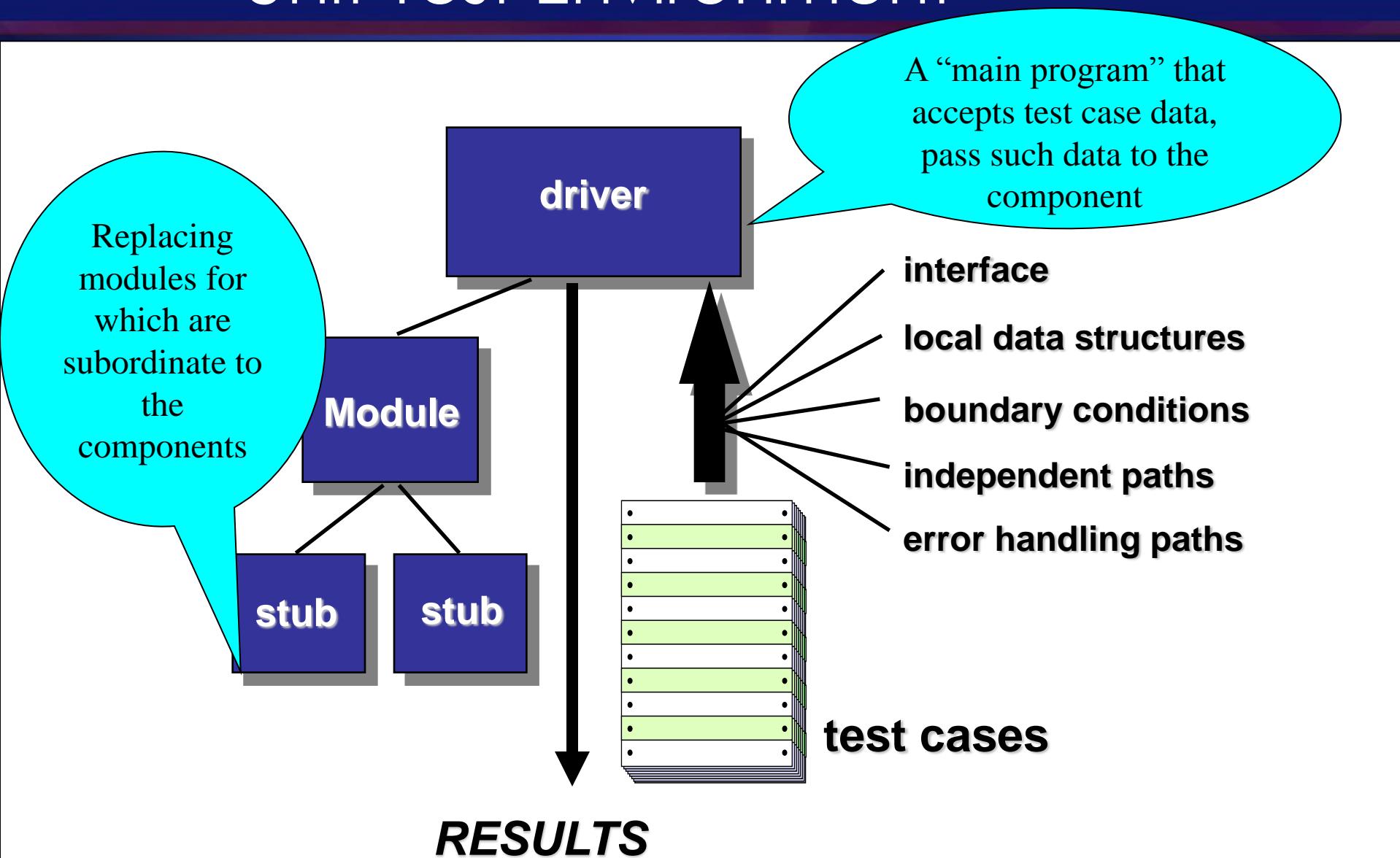
Unit Testing



Unit Testing



Unit Test Environment

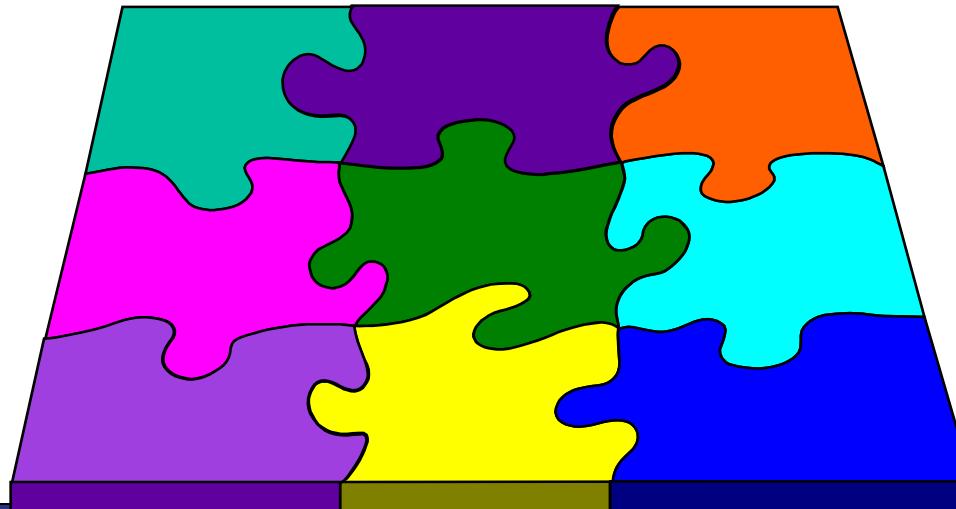


Integration Testing Strategies

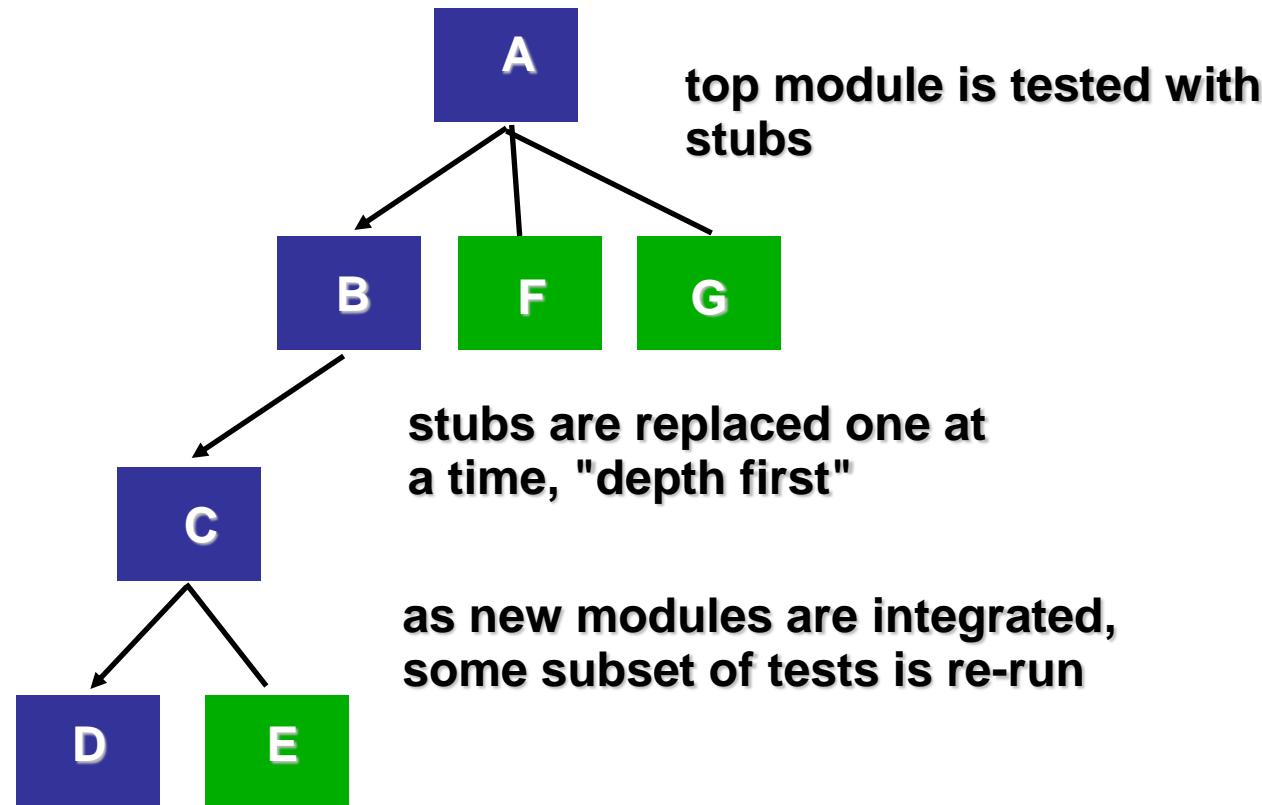
A systematic technique for constructing the SW architecture while at the same time conducting tests to uncover errors associated w/ interfacing.

Options:

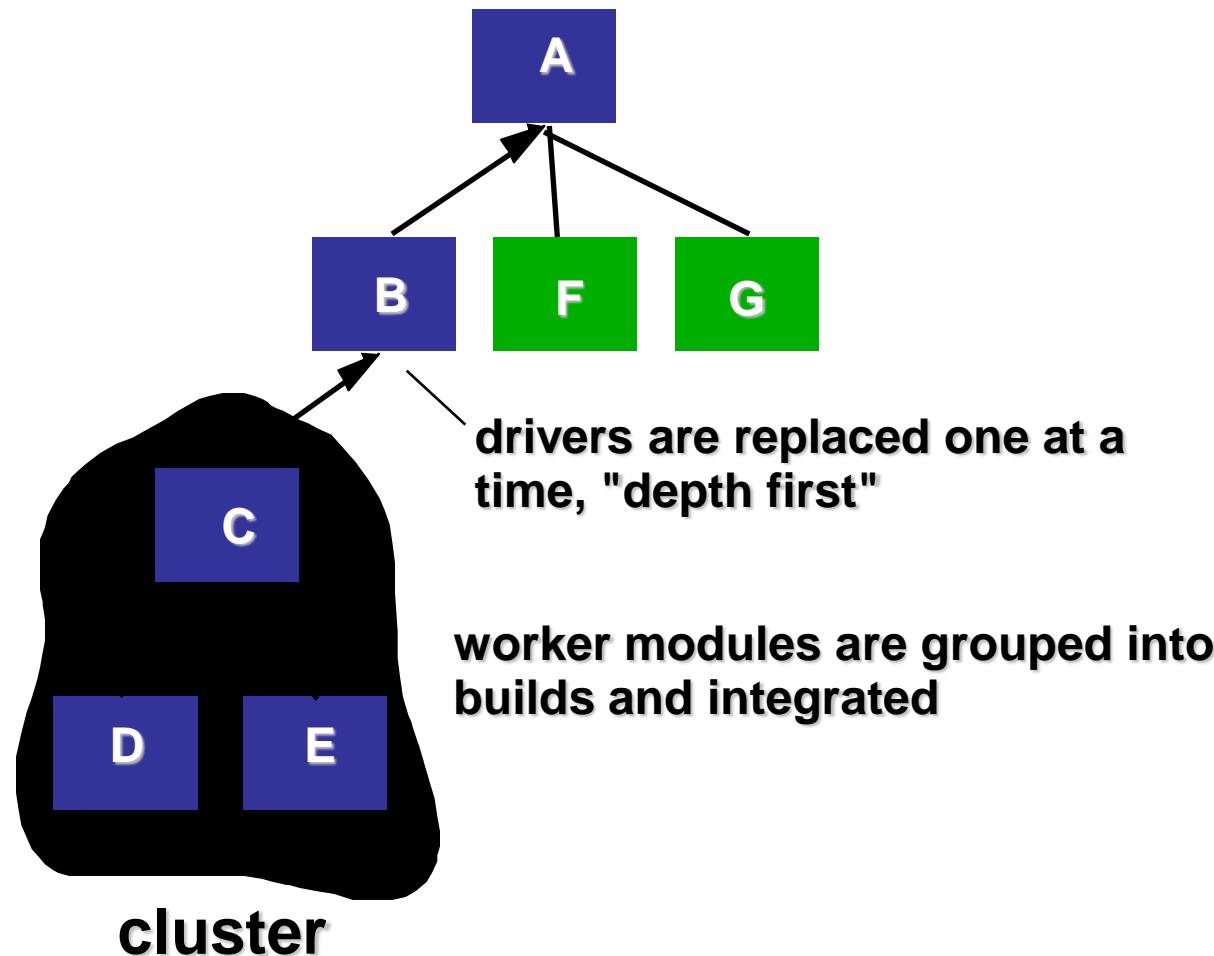
- the “big bang” approach
- an incremental construction strategy



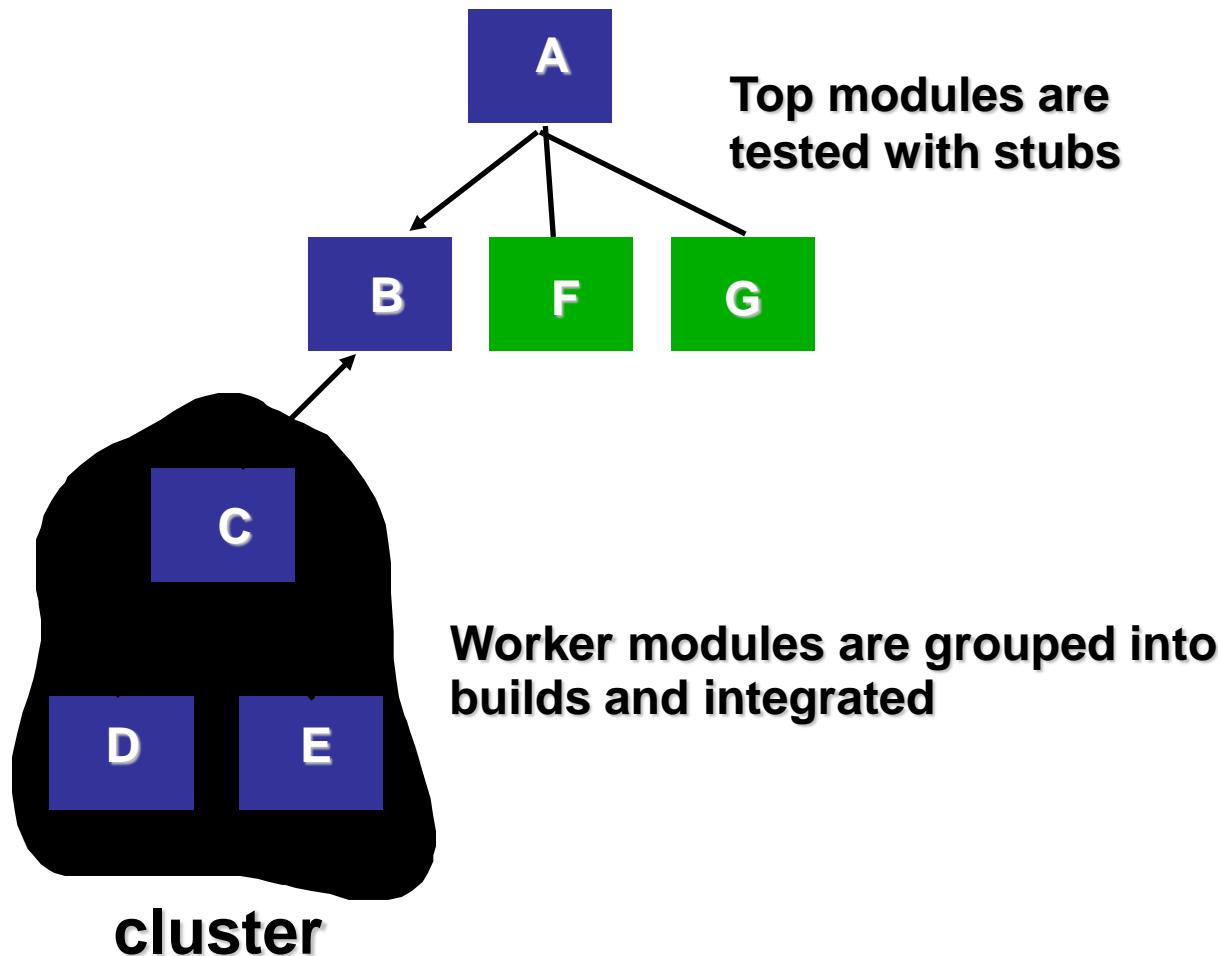
Top Down Integration



Bottom-Up Integration



Sandwich Testing



Object-Oriented Testing

- begins by evaluating the correctness and consistency of the OOAnalysis and OODesign 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

Broadening the View of “Testing”

It can be argued that the review of OO analysis and design models is especially useful because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code level. Therefore, a problem in the definition of class attributes that is uncovered during analysis will circumvent side effects that might occur if the problem were not discovered until design or code (or even the next iteration of analysis).

Testing the CRC Model

1. Revisit the CRC model and the object-relationship model.
2. Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator's definition.
3. Invert the connection to ensure that each collaborator that is asked for service is receiving requests from a reasonable source.
4. Using the inverted connections examined in step 3, determine whether other classes might be required or whether responsibilities are properly grouped among the classes.
5. Determine whether widely requested responsibilities might be combined into a single responsibility.
6. Steps 1 to 5 are applied iteratively to each class and through each evolution of the OOA model.

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

Smoke Testing

- 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
 - Integration risk is minimized.
 - The quality of the end-product is improved.
 - Error diagnosis and correction are simplified.
 - Progress is easier to access.

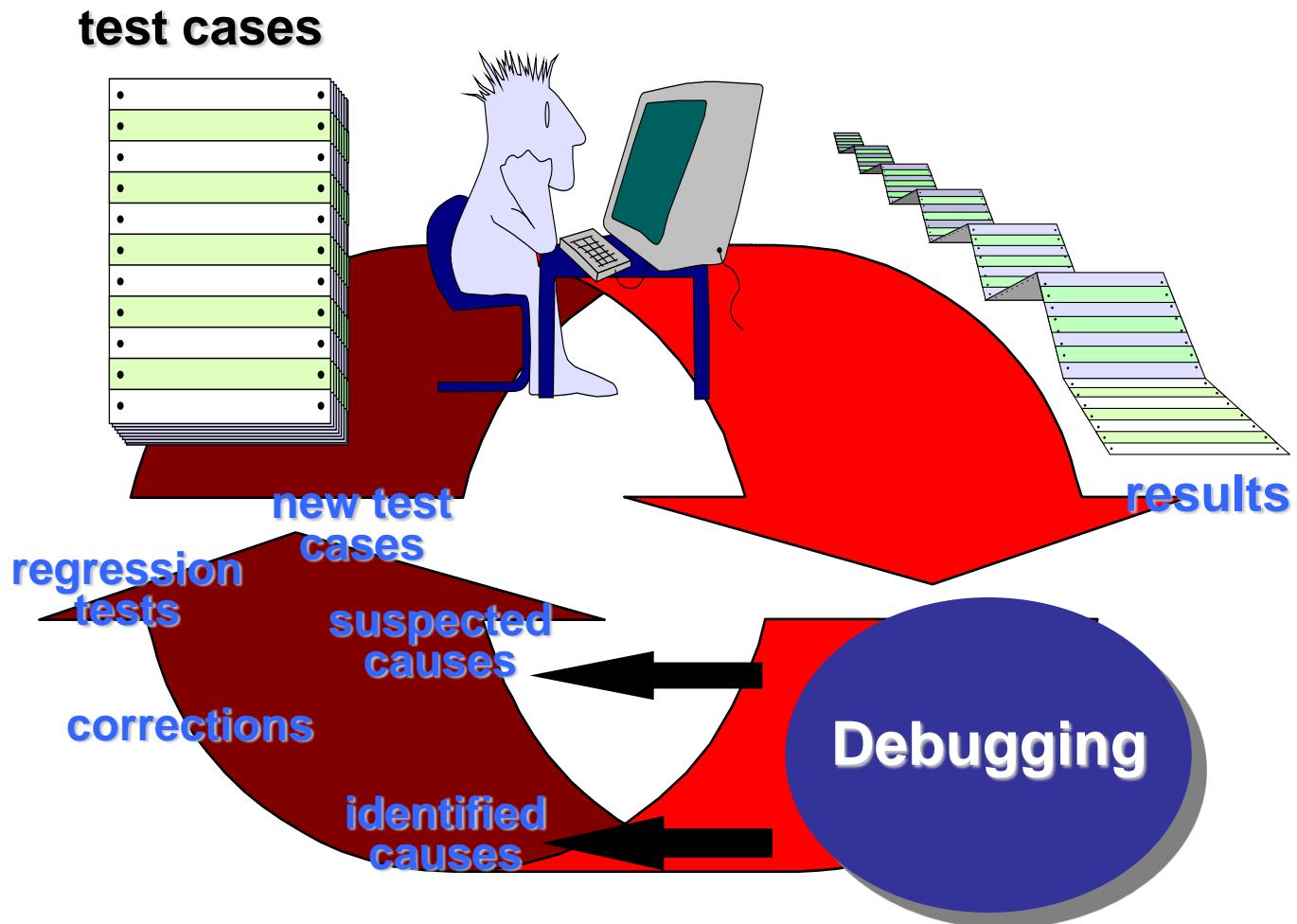
High Order Testing

- 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

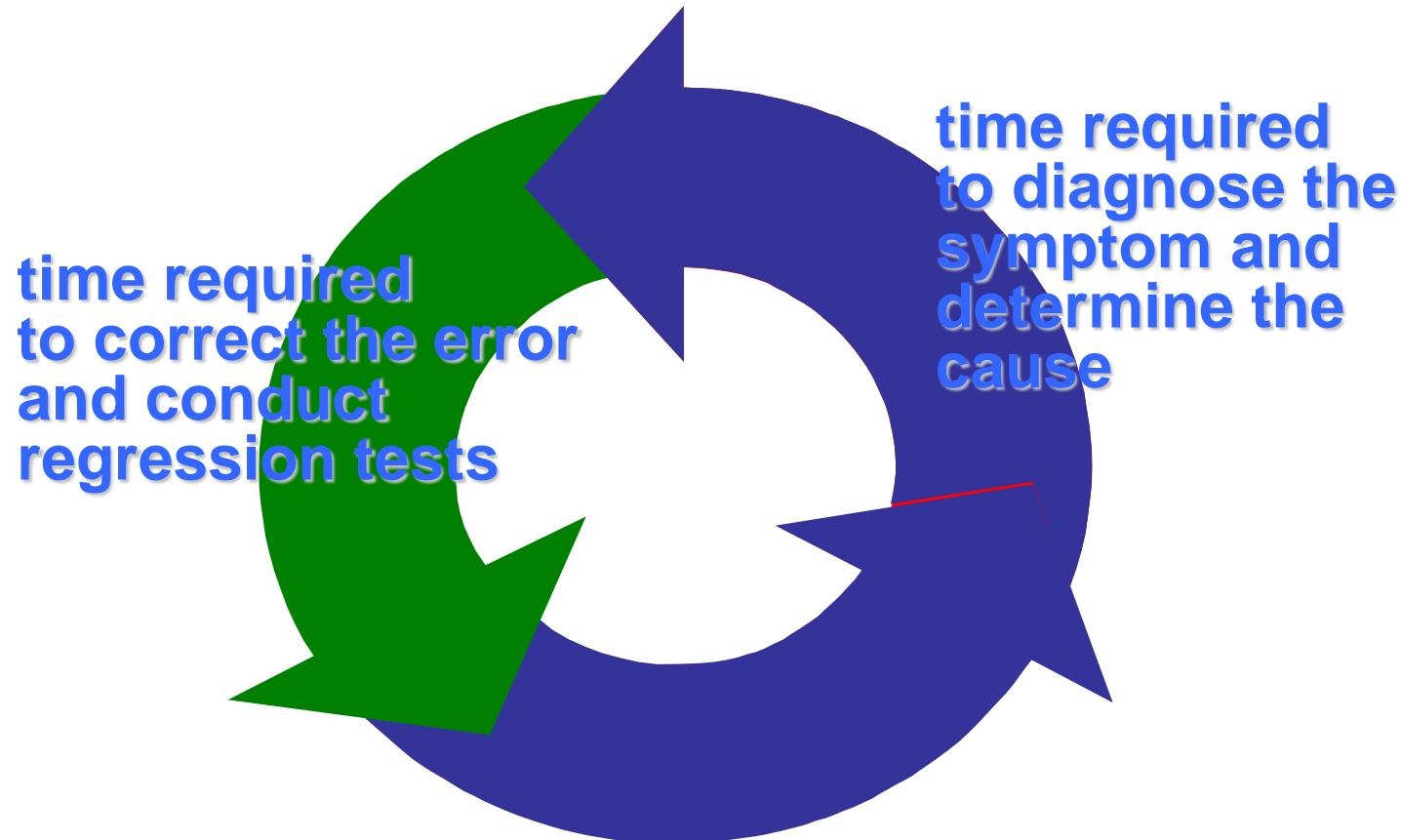
Debugging: Diagnostic Process



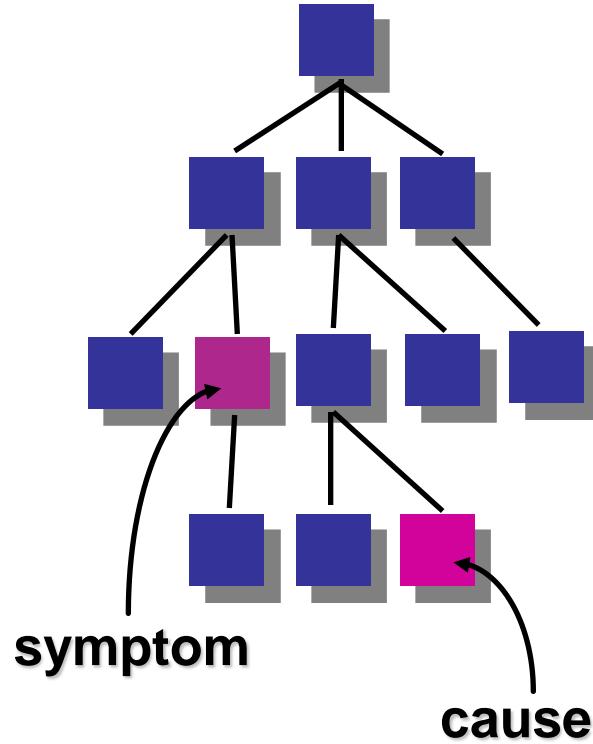
The Debugging Process



Debugging Effort

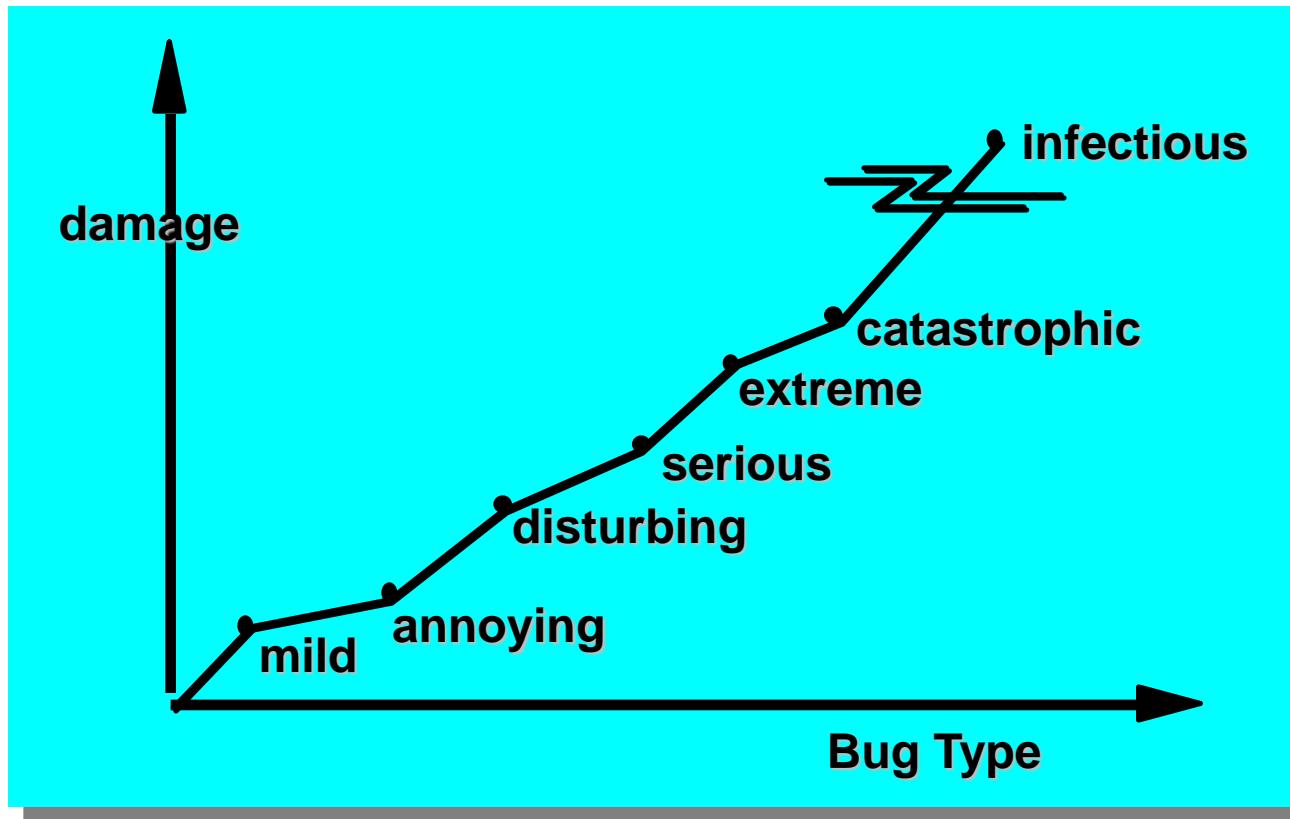


Symptoms & Causes



- 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: 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.