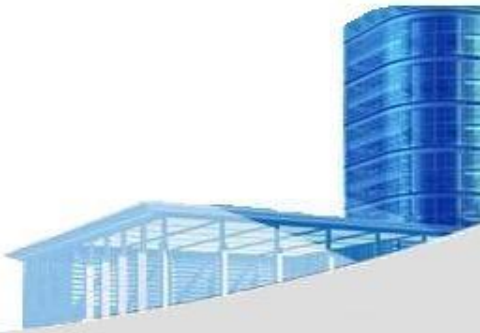
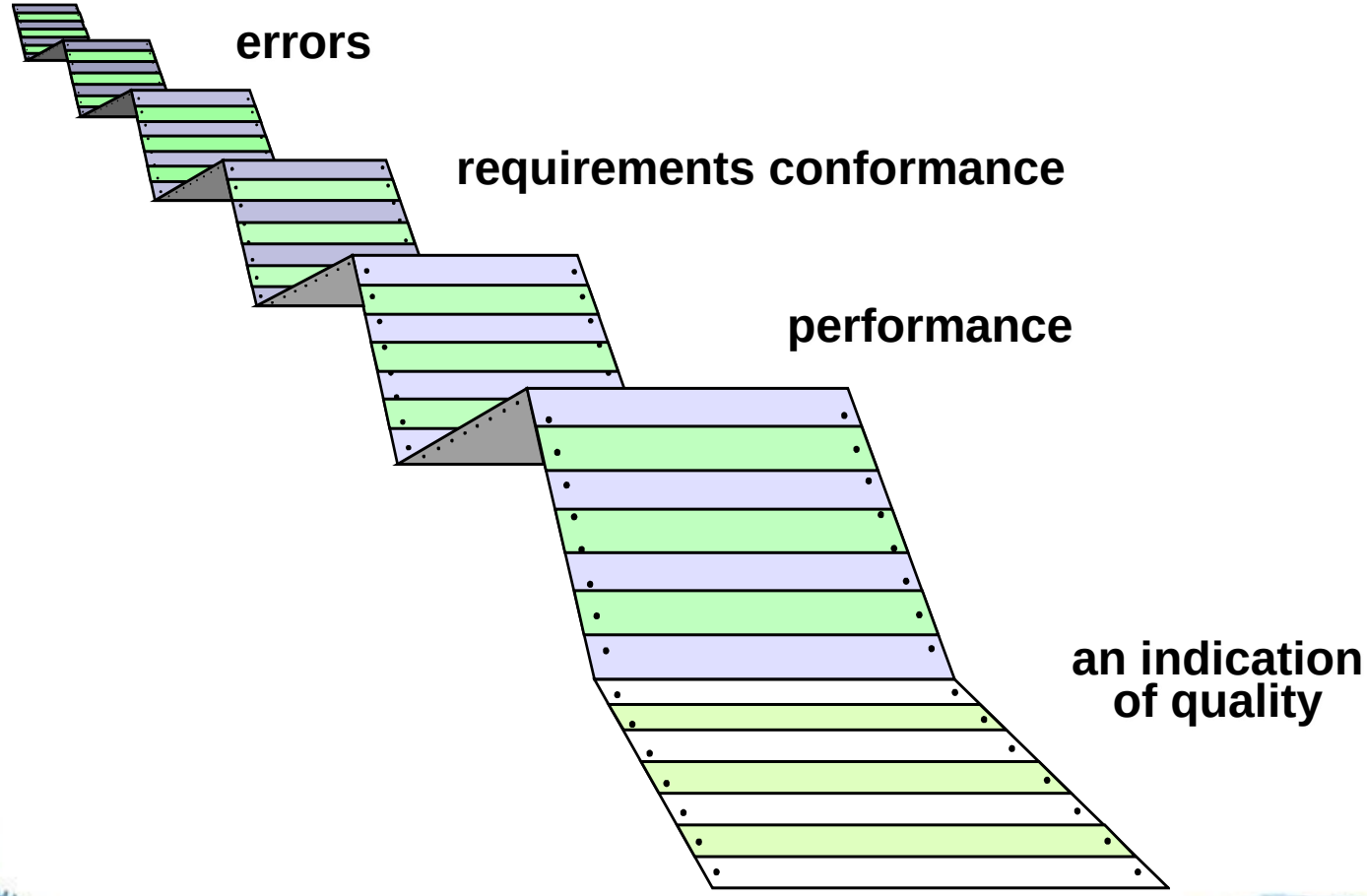
Ch.22 Software Testing Strategies



- **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**



- **Strategic Approach**

  - To perform effective testing, you should conduct effective technical reviews. By doing this, many errors will be eliminated before testing commences.
  - Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
  - Different testing techniques are appropriate for different software engineering approaches and 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.

- **V & V**

  - *Verification* refers to the set of tasks that ensure that software correctly implements a specific function.
  - *Validation* refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm [Boe81] states this another way:
    - Verification: "Are we building the product right?"
    - Validation: "Are we building the right product?"

- **Who Tests the Software?**

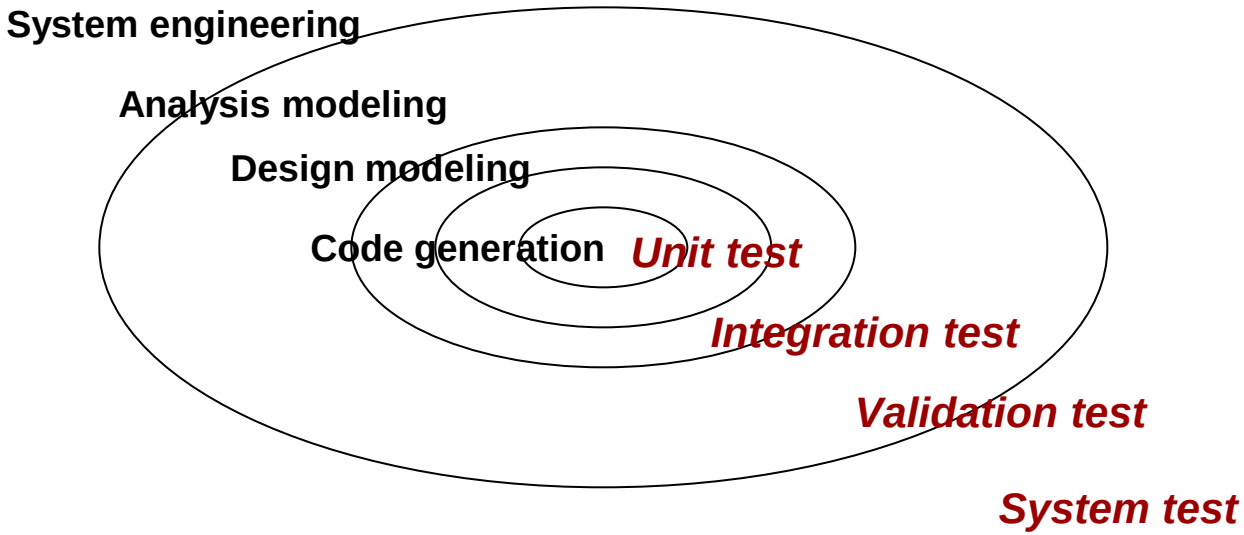
developer

independent tester

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

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

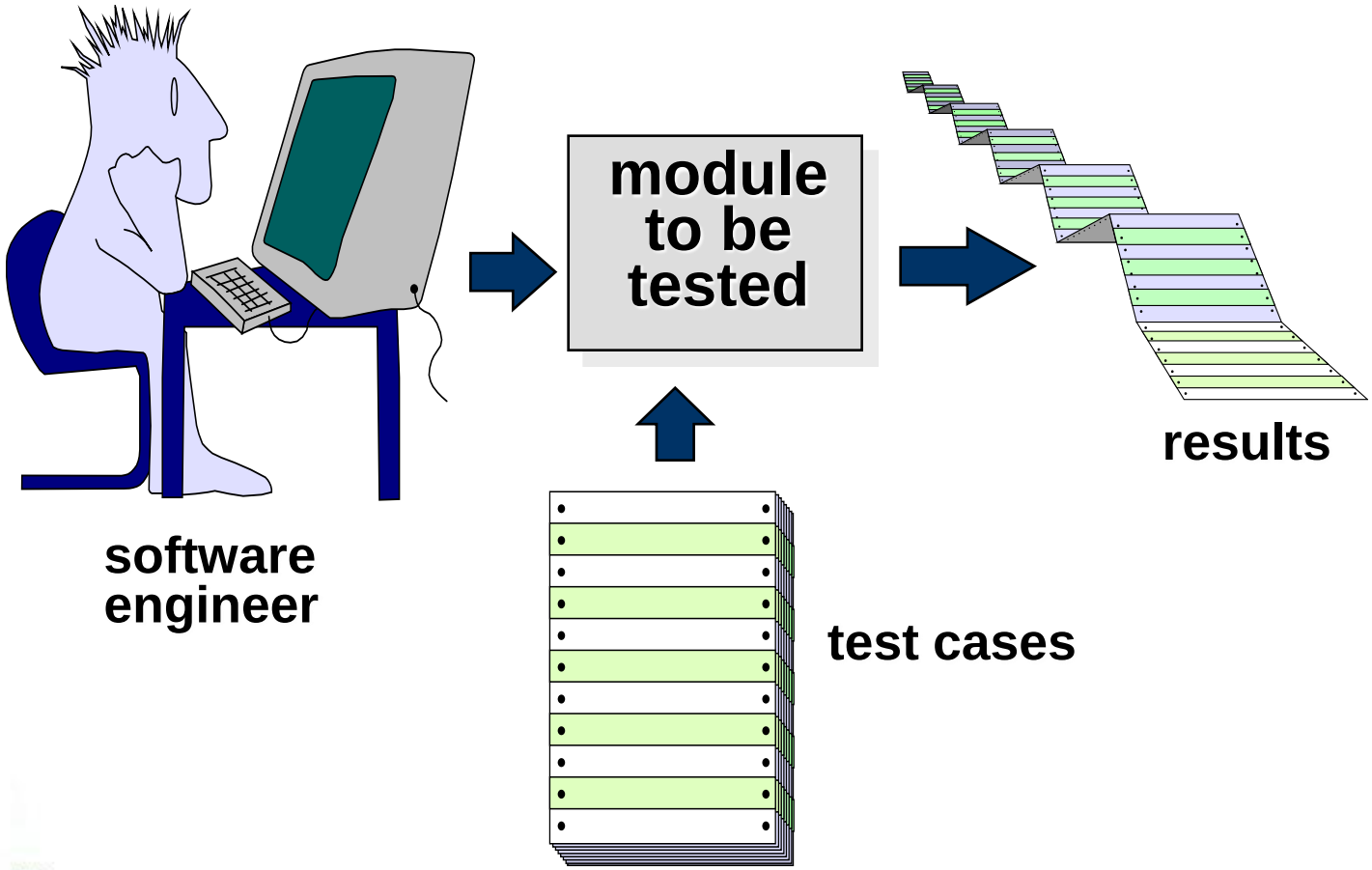
- **Testing Strategy**



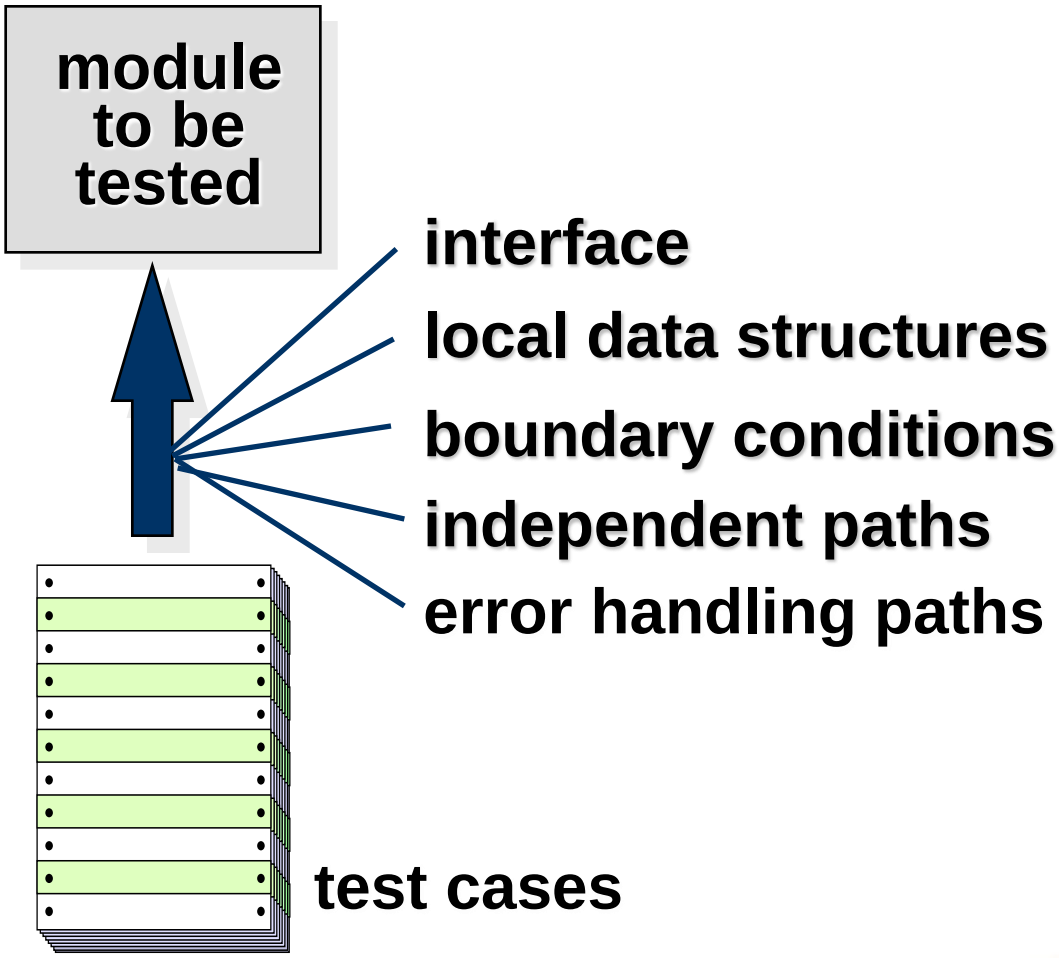
- **Testing Strategy**
  - We begin by *'testing-in-the-small'* and move toward *'testing-in-the-large'*
  - For conventional software
    - The module (component) is our initial focus
    - Integration of modules follows
  - For OO software
    - our focus when "testing in the small" changes from an 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 commences.
  - 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 technical reviews as a filter prior to testing
  - Conduct technical reviews to assess the test strategy and test cases themselves.
  - Develop a continuous improvement approach for the testing process.

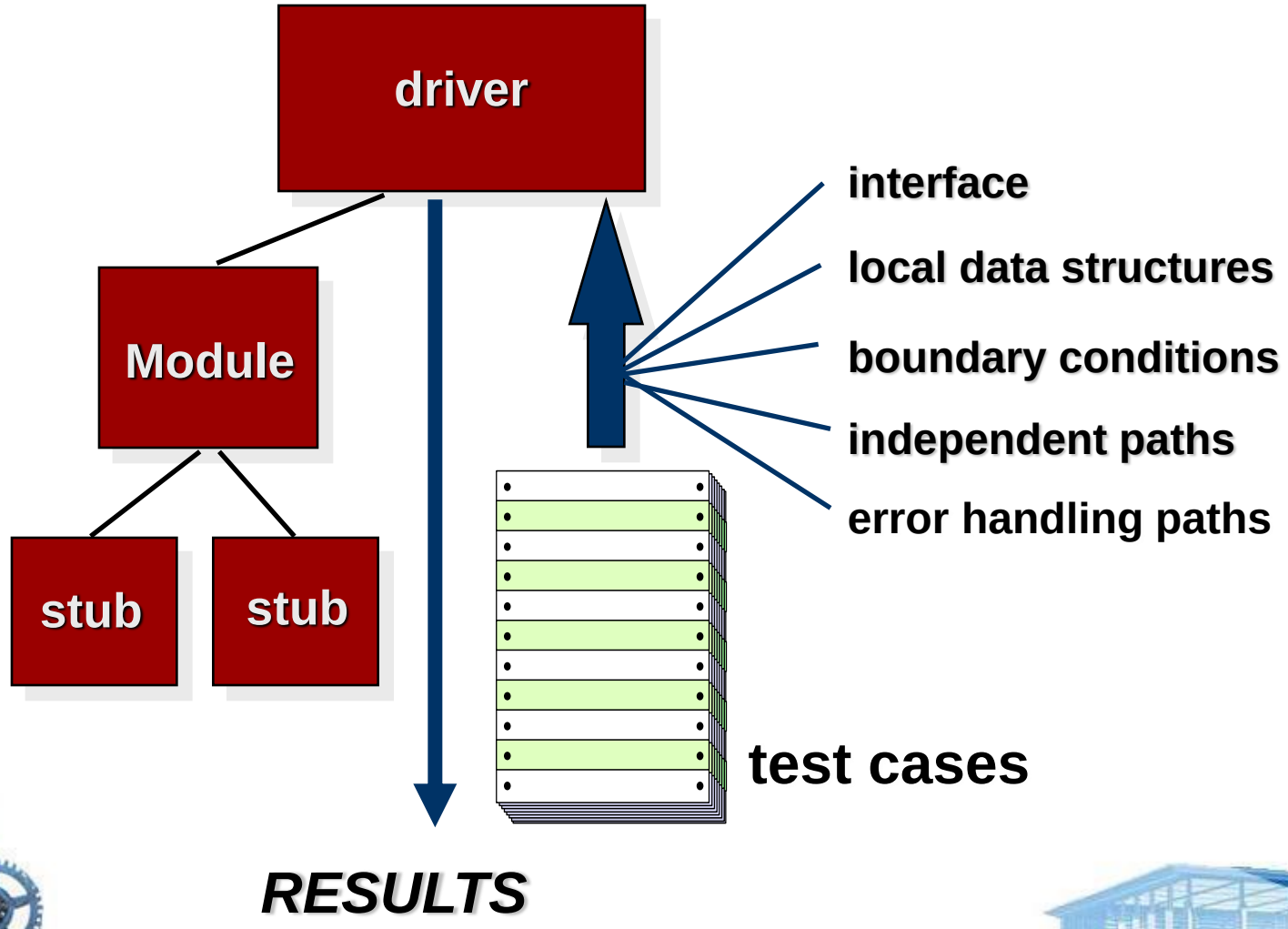
- **Unit Testing**



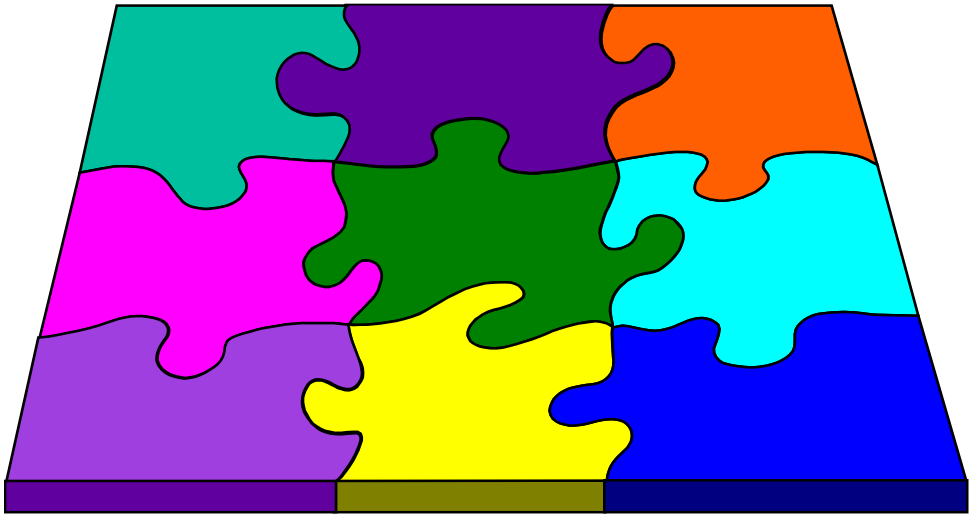
- **Unit Testing**



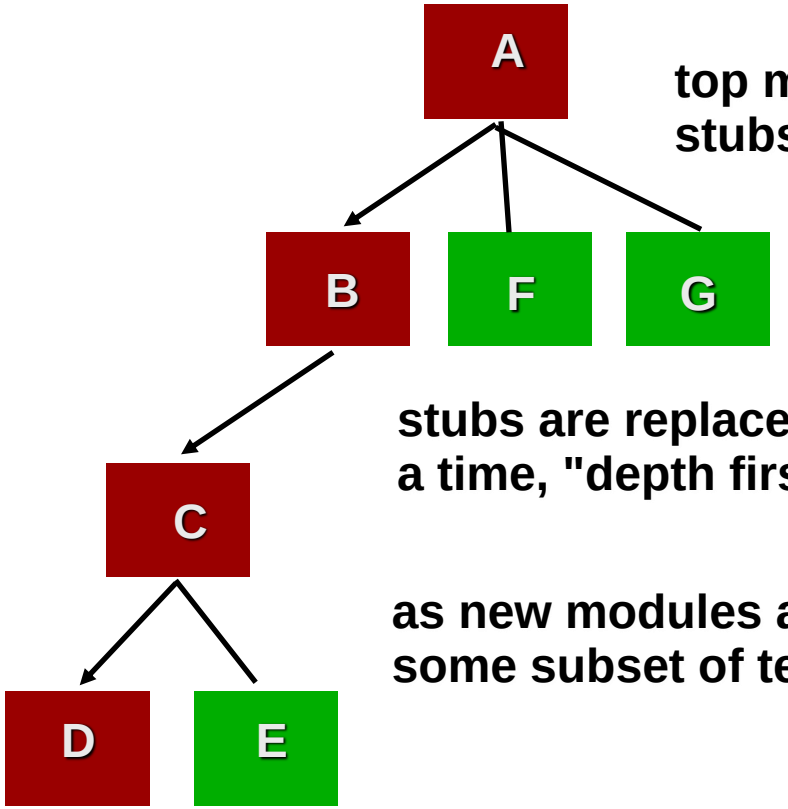
- **Unit Test Environment**



- **Integration Testing Strategies**
  - Options:
    - the “big bang” approach
    - an incremental construction strategy



- **Top Down Integration**

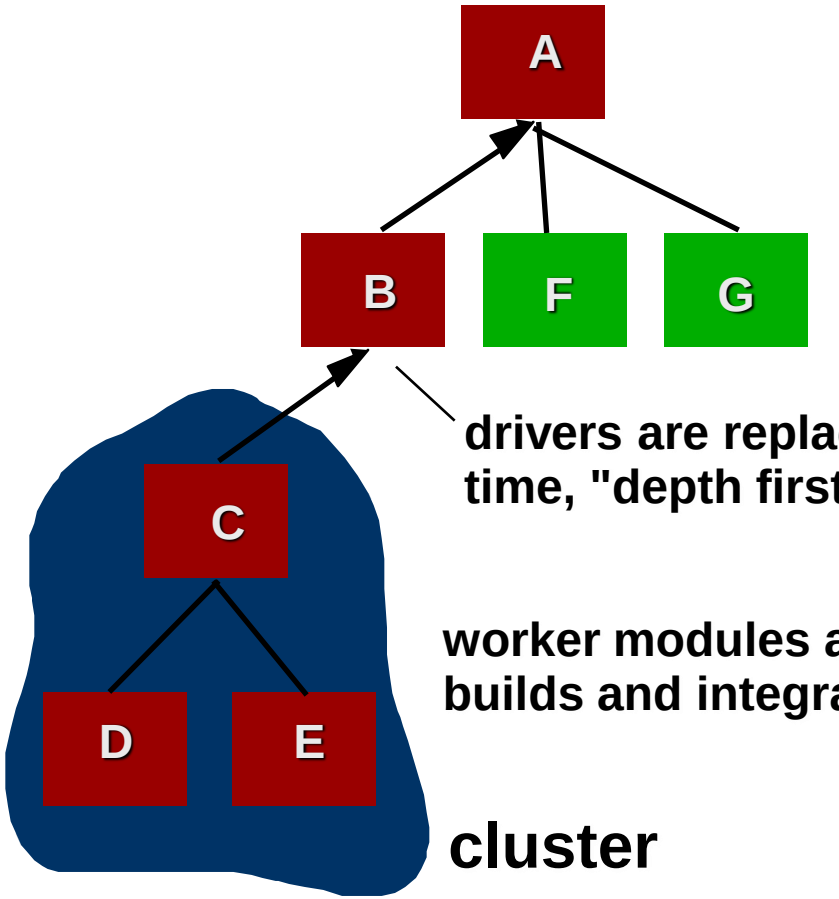


top module is tested with stubs

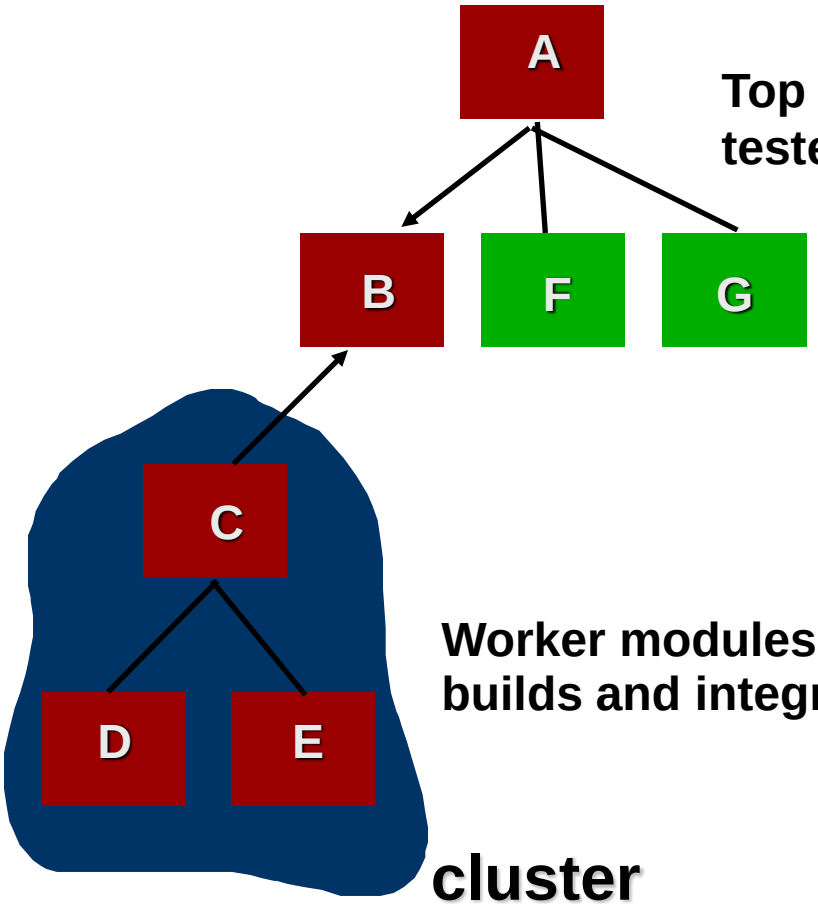
stubs are replaced one at a time, "depth first"

as new modules are integrated, some subset of tests is re-run

- **Bottom-Up Integration**



- **Sandwich Testing**



Top modules are tested with stubs

Worker modules are grouped into builds and integrated

- **Regression Testing**
  - *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.
  - Regression testing helps to ensure that changes (due to testing or for other reasons) 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.

- **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.

- **General Testing Criteria**

  - *Interface integrity* – internal and external module interfaces are tested as each module or cluster is added to the software
  - *Functional validity* – test to uncover functional defects in the software
  - *Information content* – test for errors in local or global data structures
  - *Performance* – verify specified performance bounds are tested

- **Object-Oriented Testing**
  - begins by evaluating the correctness and consistency of the analysis and design 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 analysis model.

- **OO Testing 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

- **WebApp Testing - I**
  - The content model for the WebApp is reviewed to uncover errors.
  - The interface model is reviewed to ensure that all use cases can be accommodated.
  - The design model for the WebApp is reviewed to uncover navigation errors.
  - The user interface is tested to uncover errors in presentation and/or navigation mechanics.
  - Each functional component is unit tested.

- **WebApp Testing - II**
  - Navigation throughout the architecture is tested.
  - The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.
  - Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.
  - Performance tests are conducted.
  - The WebApp is tested by a controlled and monitored population of end-users. The results of their interaction with the system are evaluated for content and navigation errors, usability concerns, compatibility concerns, and WebApp reliability and performance.

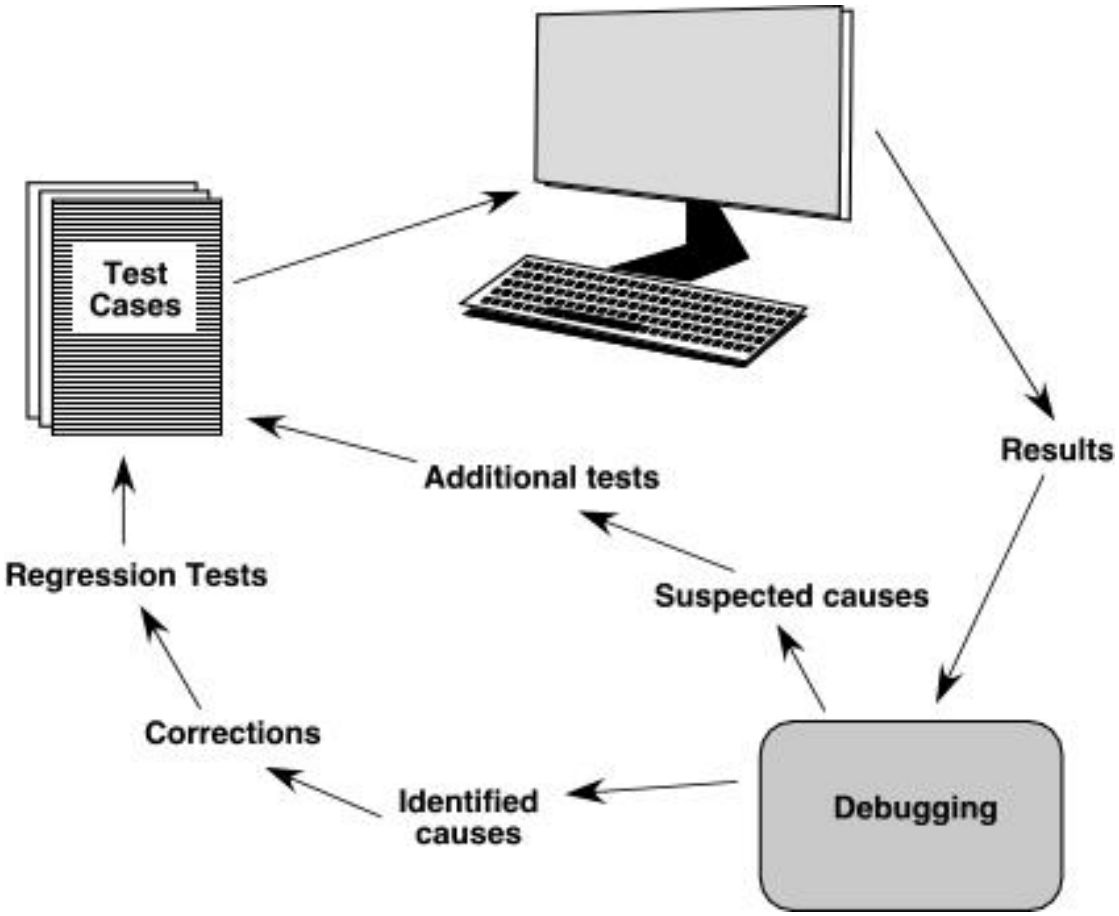
- **MobileApp Testing**

  - *User experience testing* – ensuring app meets stakeholder usability and accessibility expectations
  - *Device compatibility testing* – testing on multiple devices
  - *Performance testing* – testing non-functional requirements
  - *Connectivity testing* – testing ability of app to connect reliably
  - *Security testing* – ensuring app meets stakeholder security expectations
  - *Testing-in-the-wild* – testing app on user devices in actual user environments
  - *Certification testing* – app meets the distribution standards

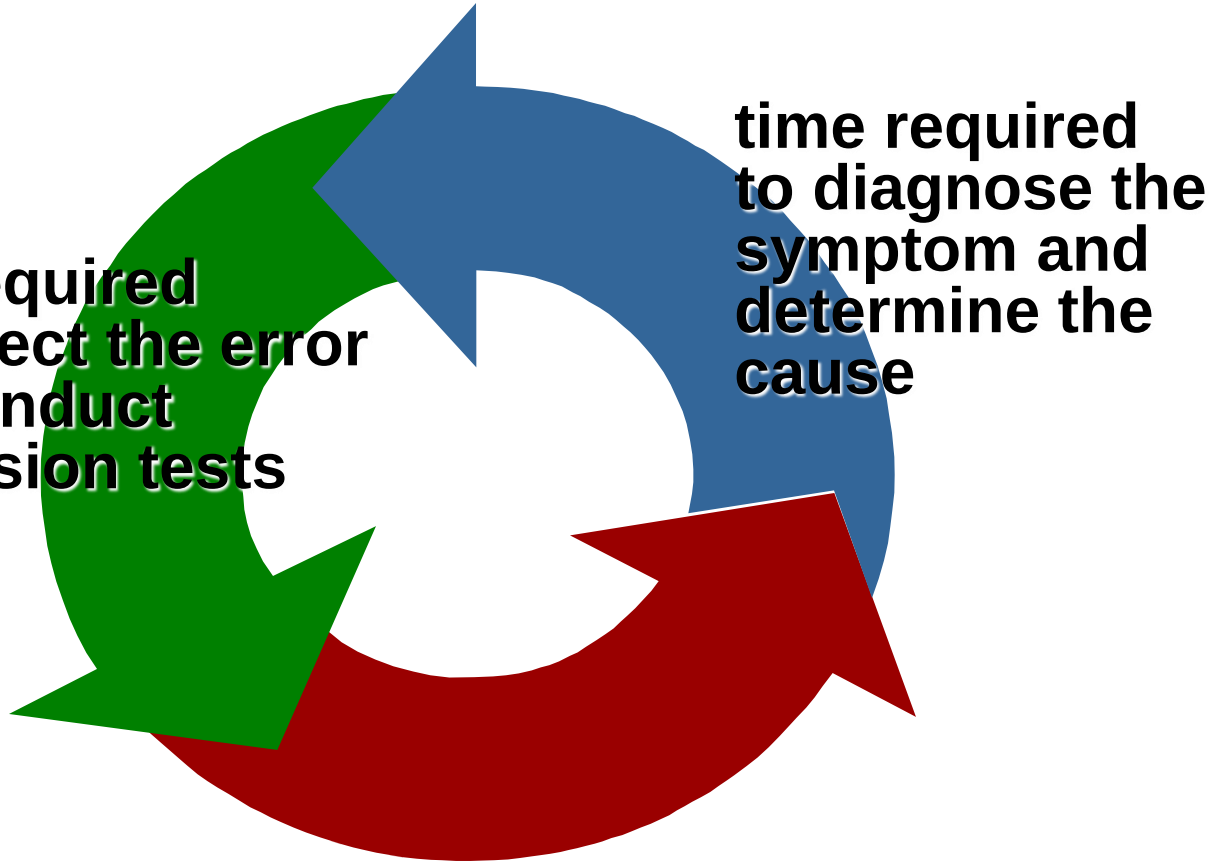
- **High Order Testing**
  - *Validation testing*
    - Focus is on software requirements
  - *System testing*
    - Focus is on system integration
  - *Alpha/Beta testing*
    - Focus is on customer usage
  - *Recovery testing*
    - forces the software to fail in a variety of ways and verifies that 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: A Diagnostic Process**

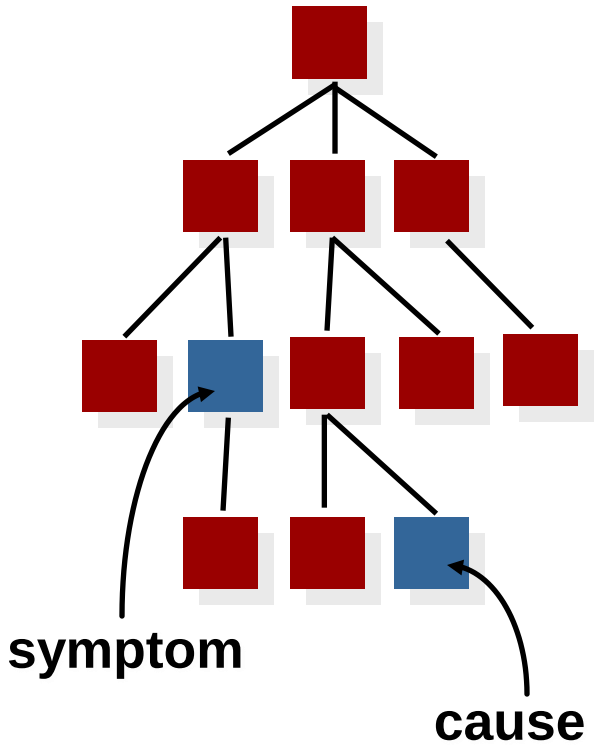
- **The Debugging Process**



- **Debugging Effort**

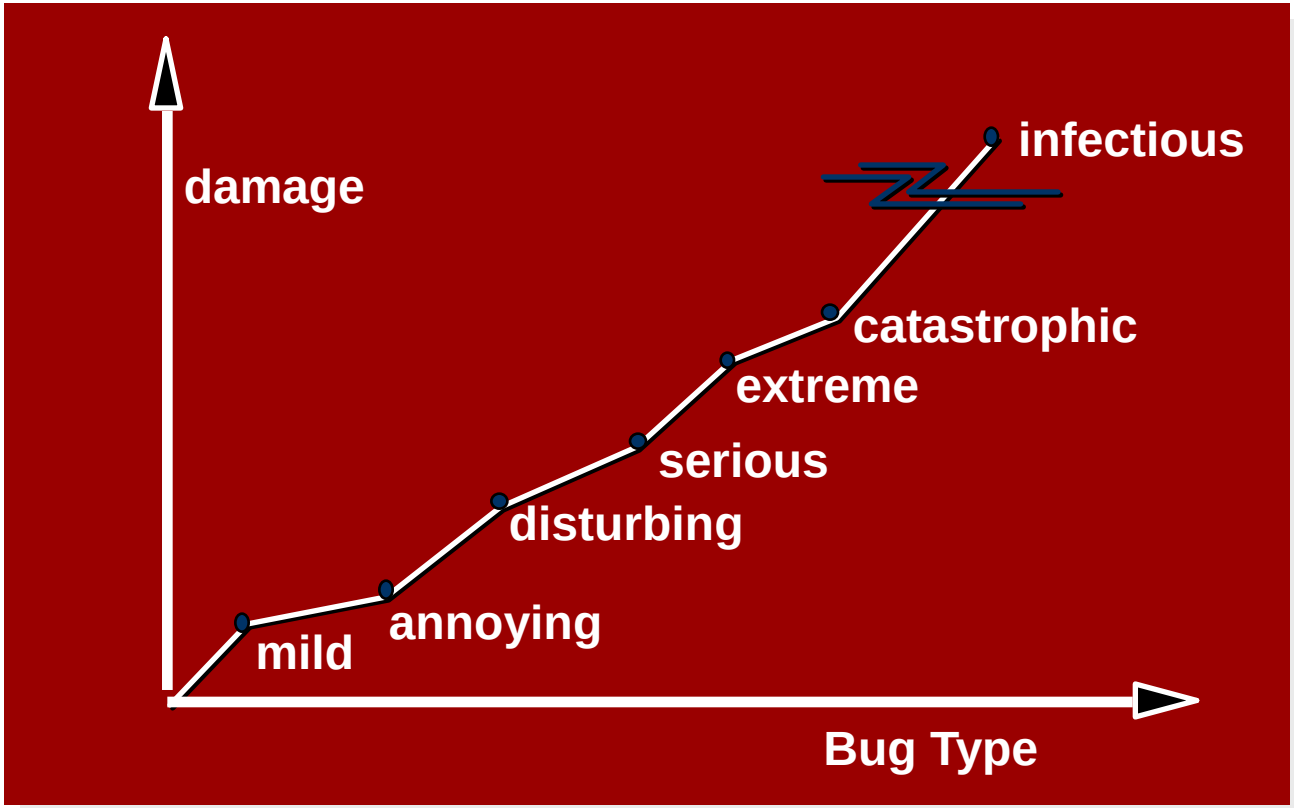


- **Symptoms & Causes**



- 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

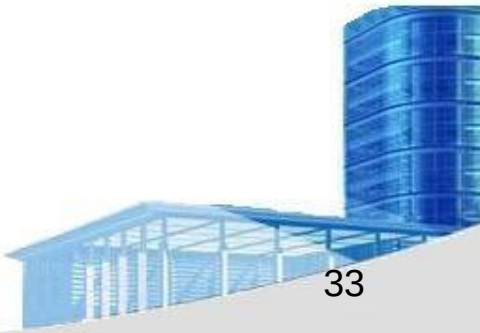
- **Consequences of Bugs**



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



- **Correcting the Error**
  - *Is the cause of the bug reproduced in another part of the program?* In many situations, a program defect is caused by an erroneous pattern of logic that may be reproduced elsewhere.
  - *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.
  - *What could we have done to prevent this bug in the first place?* This question is the first step toward establishing a statistical software quality assurance approach. If you correct the process as well as the product, the bug will be removed from the current program and may be eliminated from all future programs.

- **Final Thoughts**

  - Think -- before you act to correct
  - Use tools to gain additional insight
  - If you're at an impasse, get help from someone else
  - Once you correct the bug, use regression testing to uncover any side effects