# Software Testing Strategies

# Introduction

- A strategy for software testing integrates the design of software test cases into a well-planned series of steps that result in successful development of the software
- The strategy provides a road map that describes the steps to be taken, when, and how much effort, time, and resources will be required
- The strategy incorporates test planning, test case design, test execution, and test result collection and evaluation
- The strategy provides guidance for the practitioner and a set of milestones for the manager
- Because of time pressures, progress must be measurable and problems must surface as early as possible

# Software Testing

- A Strategic Approach
  - Testing is a set of activities that can be planned in advance and conducted systematically
    - Conduct effective formal technical reviews
    - Begin at the component level and work “outward” toward the integration of the entire computer-based system
    - Different testing techniques at different points in time
    - Usually conducted by the software developer and for large projects by an independent test group
    - Testing and debugging are different activities, but debugging must be accommodated in any testing strategy

# Software Testing

- Verification and Validation
  - Software testing is often referred to as V&V
    - Verification refers to the set of activities that ensures that software correctly implements a specific function
      - Are we building the product right?
    - Validation is a different set of activities that ensure that the software that has been built is traceable to customer requirements
      - Are we building the right product?

# Software Testing

- Verification and Validation
  - Encompass a wide array of Software Quality Assurance (SQA) activities including
    - Formal technical reviews
    - Quality and configuration audits
    - Performance monitoring
    - Simulation
    - Feasibility study
    - Documentation review
    - Database review
    - Algorithm analysis
    - Development, usability, installation testing

# Software Testing

- Organizing for Software Testing
  - Psychological point of view
    - The software engineer analyzes, models, and then creates a computer program and its documentation
      - From the software engineers perspective testing is destructive as it tries to break the thing that SE has built
    - There are often a number of misconceptions
      - The developer of the software should no do test
      - Software should be tossed over the wall to strangers who will test it mercilessly
      - Testers get involved with the project only when the testing steps are about to begin

# Software Testing

- Organizing for Software Testing
  - The role of the independent test group
    - Remove the inherent problems associated with letting the builder test the thing that has been built
    - Removes the conflict of interest
    - The software engineer, however, does turn the program over to the ITG and walk away; both have to work closely throughout the project
    - The developer must be available to correct uncovered errors

# Software Testing

- Strategy for conventional software architecture
  - Unit Testing
    - Concentrates on each unit (i.e. component) of the software as implemented in the source code
  - Integration Testing
    - Focus is on design and the construction of the software architecture
  - Validation Testing
    - Requirements are analysis are validated against the software that has been constructed
  - System Testing
    - Software and other elements are tested as a whole

# 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

# Software Testing

- Strategic Issues
  - Specify product requirements in a quantifiable manner long before testing
    - A good strategy not only looks for errors, but also asses other quality characteristics such as
      - Portability
      - Maintainability
      - Usability
  - State testing objectives explicitly
    - The specific objectives of testing should be stated in measurable terms
      - Test effectiveness
      - The cost to find and fix defects
      - Test coverage

# Software Testing

- Strategic Issues
  - Understand the user of the software and develop a profile for each
    - 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 project
  - Build robust software that is designed to test itself
    - Software should be designed in a manner that uses anti-bugging techniques. Software should be capable of diagnosing certain classes of errors. The design should be accommodated automated testing and regression testing

# Software Testing

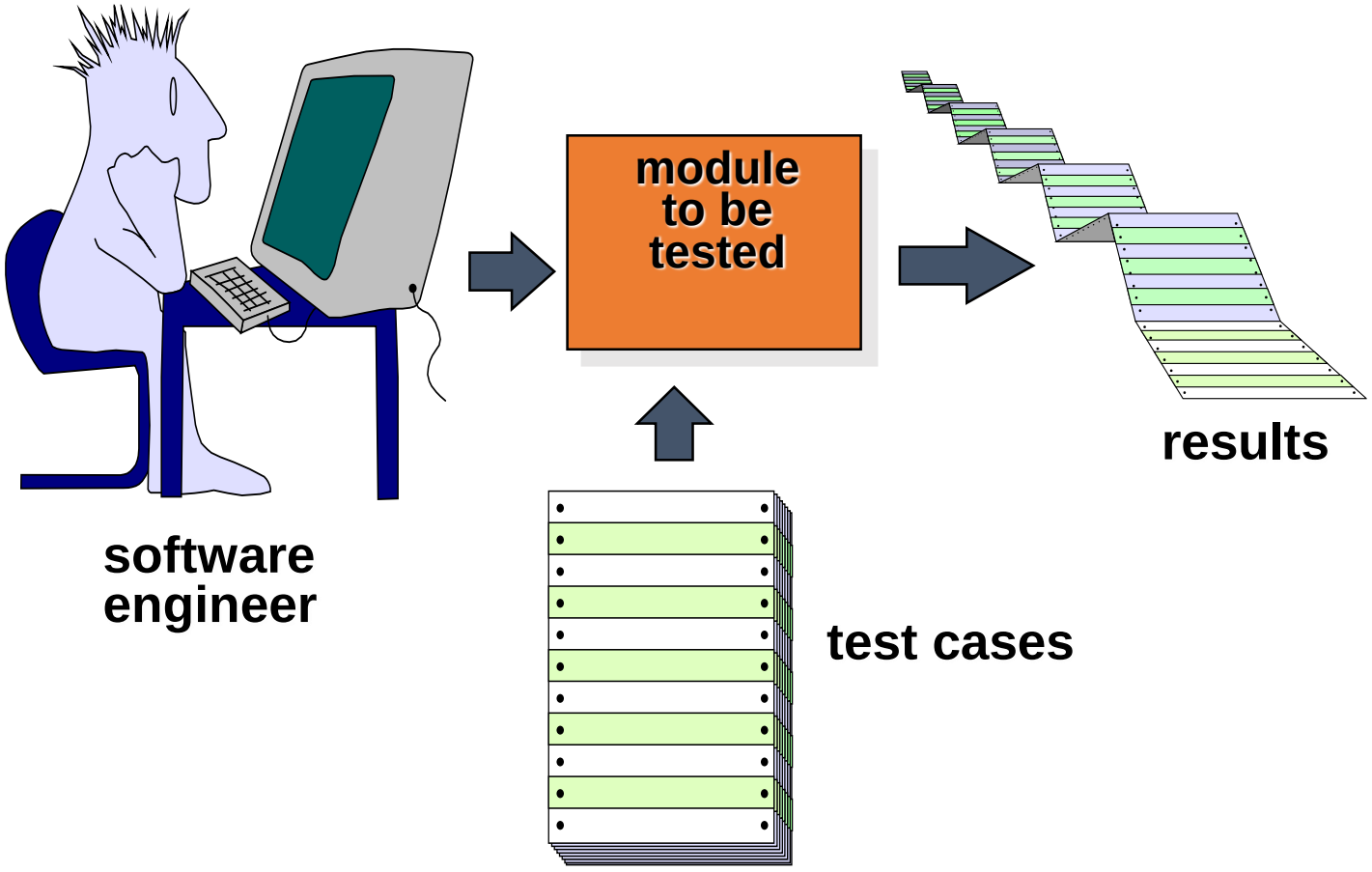
- Strategic Issues
  - Use effective formal technical reviews as a filter prior to testing
    - Reviews can reduce the amount of testing effort that is required to produce high-quality software
  - Develop a continuous improvement approach
    - The testing strategy should be measured. The metrics collected during testing should be used as part of a statistical process control approach for testing

# Test Strategies for Conventional Software

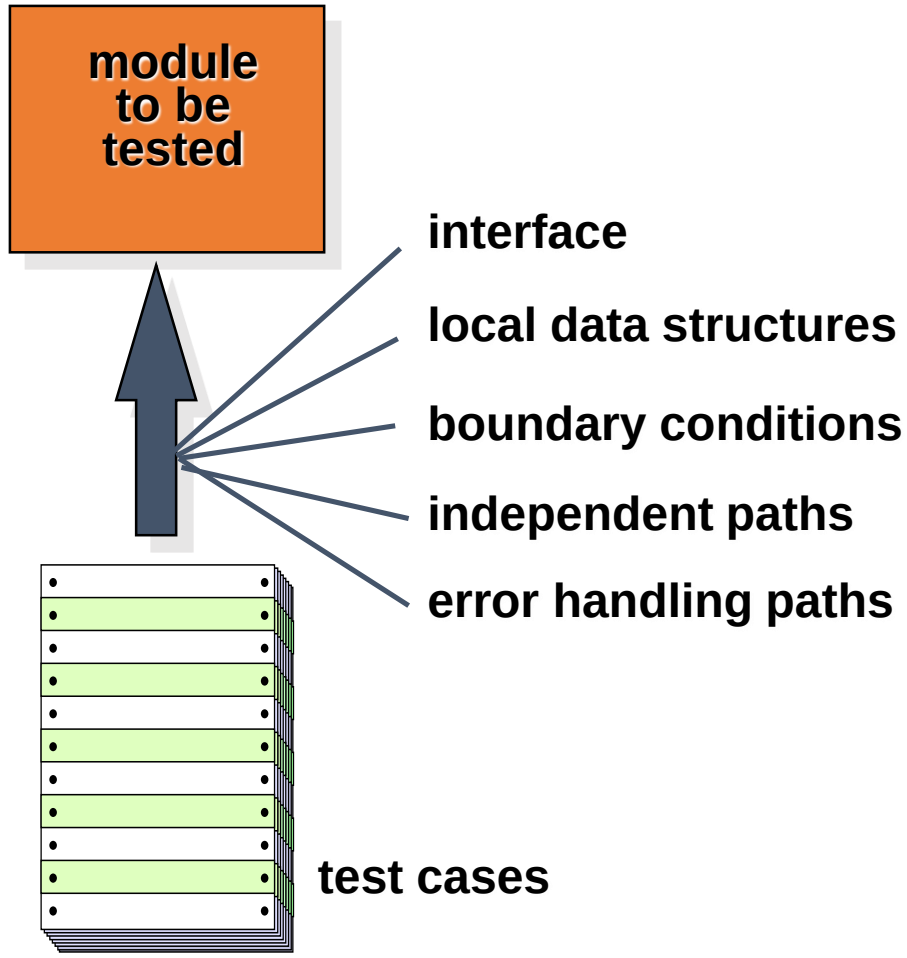
# Unit Testing

- 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



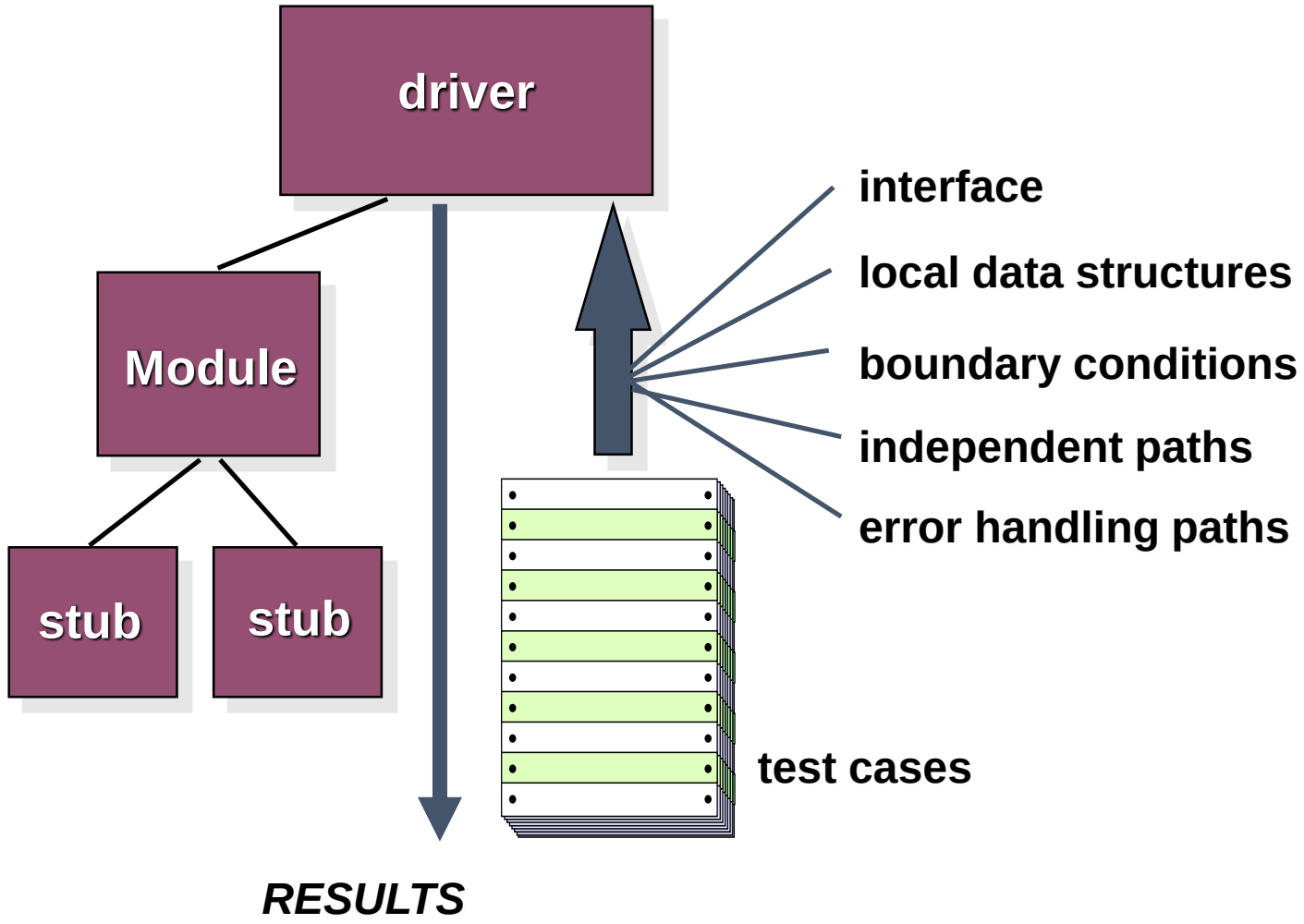
# Unit Testing



# 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

# Unit Test Environment



# 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

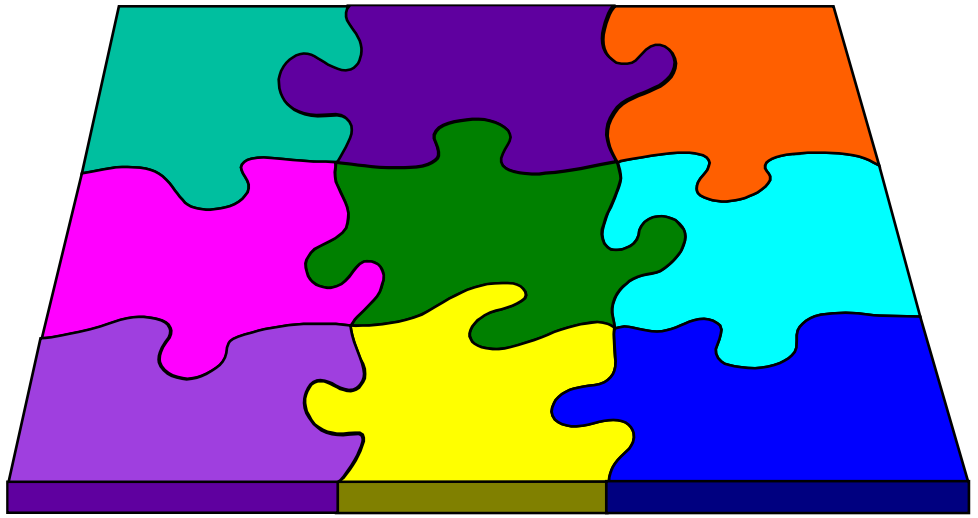
# Integration Testing

- 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**



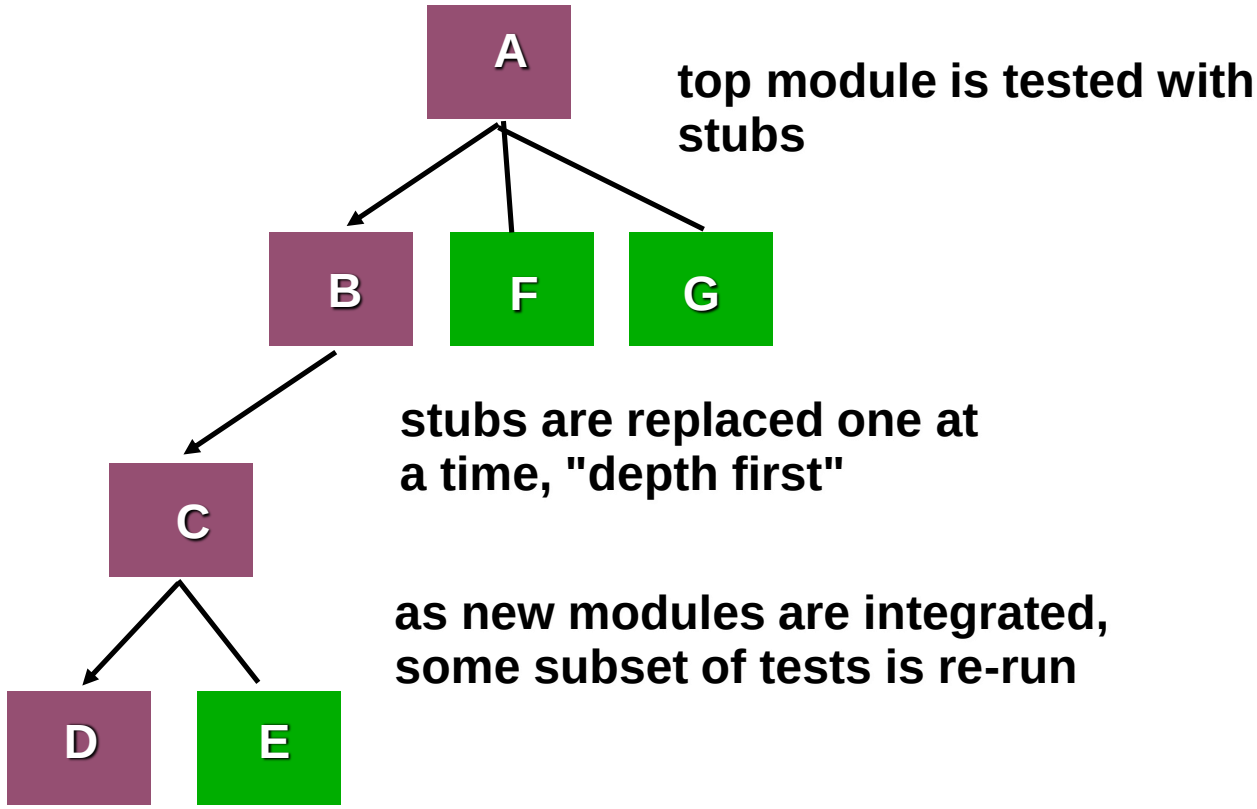
# Incremental Integration Testing

- 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

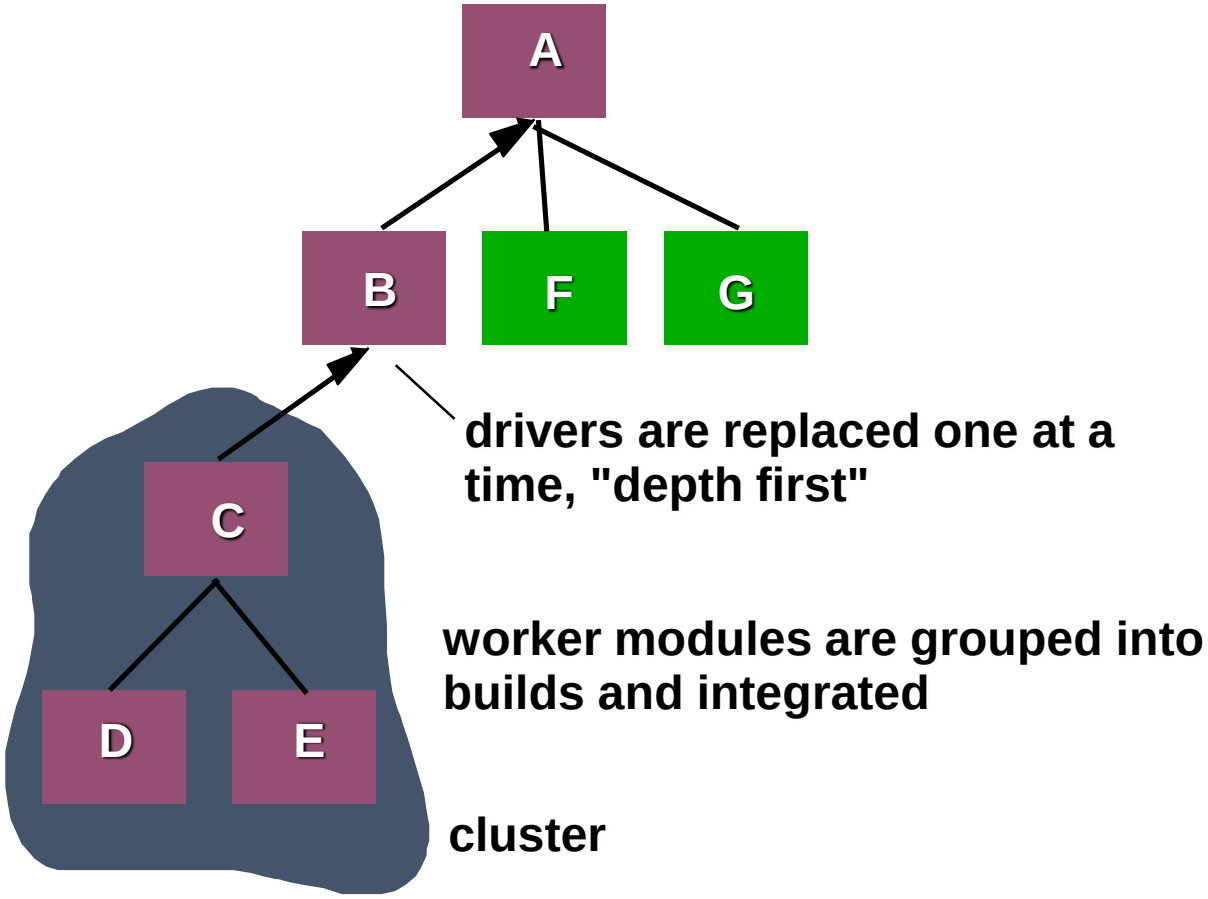
# Top Down Integration



# 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

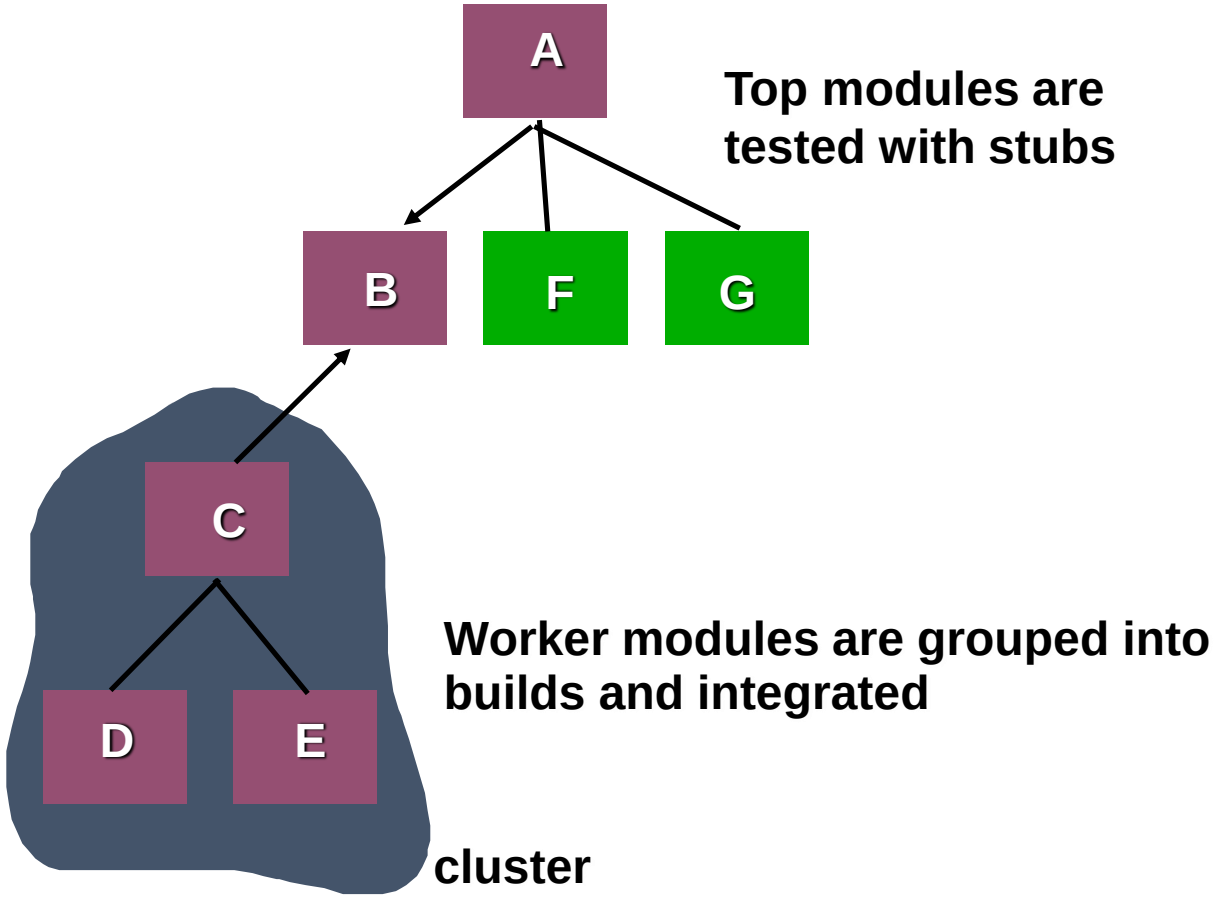
# Bottom-Up Integration



# 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



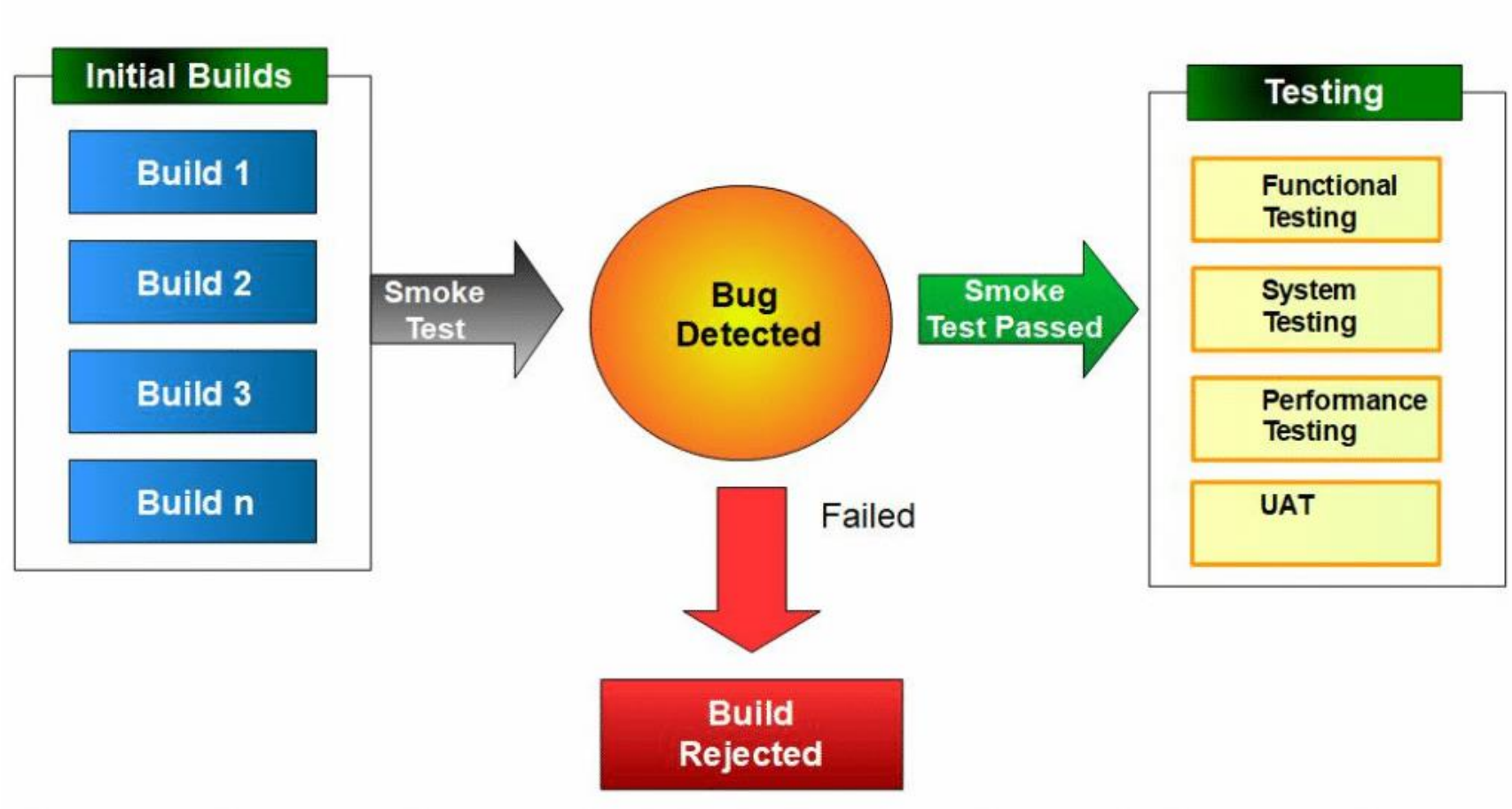
# Regression Testing

- *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.

# Smoke Testing (cont'd)



# 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

# Test Strategies for Object-Oriented Software

- With object-oriented software, you can no longer test a single operation in isolation (conventional thinking)
- Traditional top-down or bottom-up integration testing has little meaning
- Class testing for object-oriented software is the equivalent of unit testing for conventional software
  - Focuses on operations encapsulated by the class and the state behavior of the class
- Drivers can be used
  - To test operations at the lowest level and for testing whole groups of classes
  - To replace the user interface so that tests of system functionality can be conducted prior to implementation of the actual interface
- Stubs can be used
  - In situations in which collaboration between classes is required but one or more of the collaborating classes has not yet been fully implemented

# Test Strategies for Object-Oriented Software (continued)

- Two different object-oriented testing strategies
  - Thread-based testing
    - Integrates the set of classes required to respond to one input or event for the system
    - Each thread is integrated and tested individually
    - Regression testing is applied to ensure that no side effects occur
  - Use-based testing
    - First tests the independent classes that use very few, if any, server classes
    - Then the next layer of classes, called dependent classes, are integrated
    - This sequence of testing layer of dependent classes continues until the entire system is constructed

# Background of validation testing

- 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**

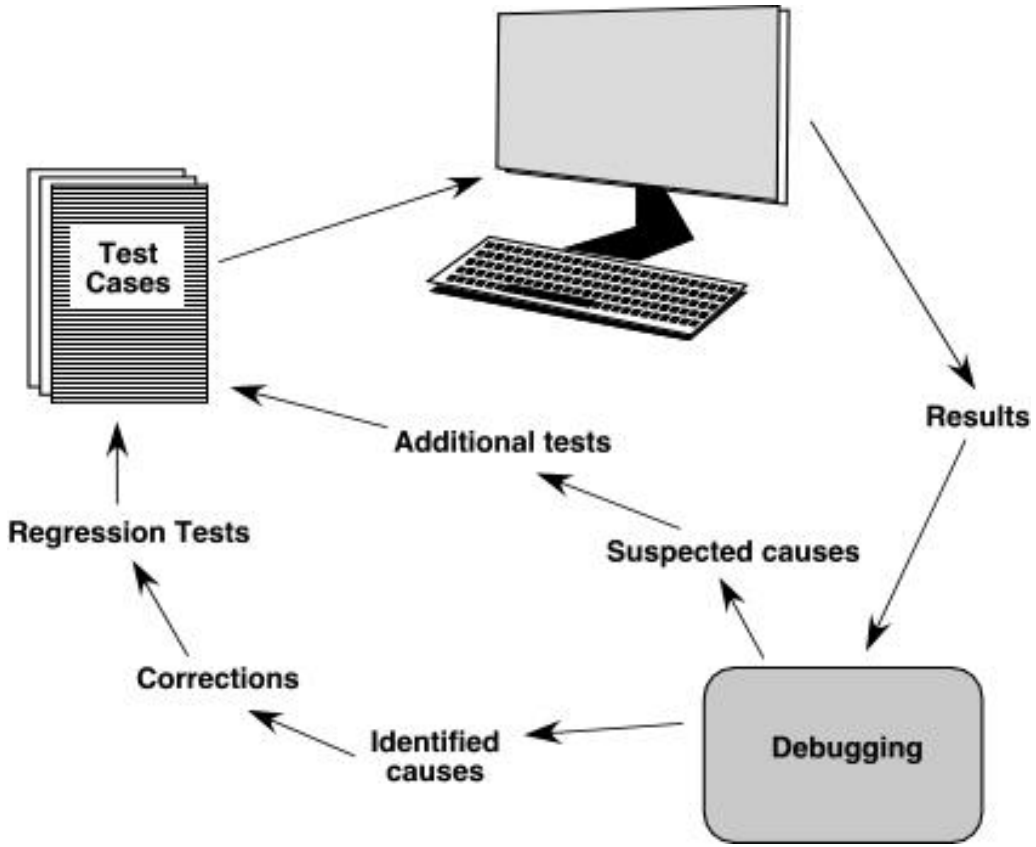
- Tests the run-time performance of software within the context of an integrated system
- Often coupled with stress testing and usually requires both hardware and software instrumentation
- Can uncover situations that lead to degradation and possible system failure

# The Art of Debugging

# Debugging 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 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

# Thank you