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.

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

Strategic Approach

To perform effective testing, conduct effective technical reviews. By doing this, many errors will be eliminated before testing commences.

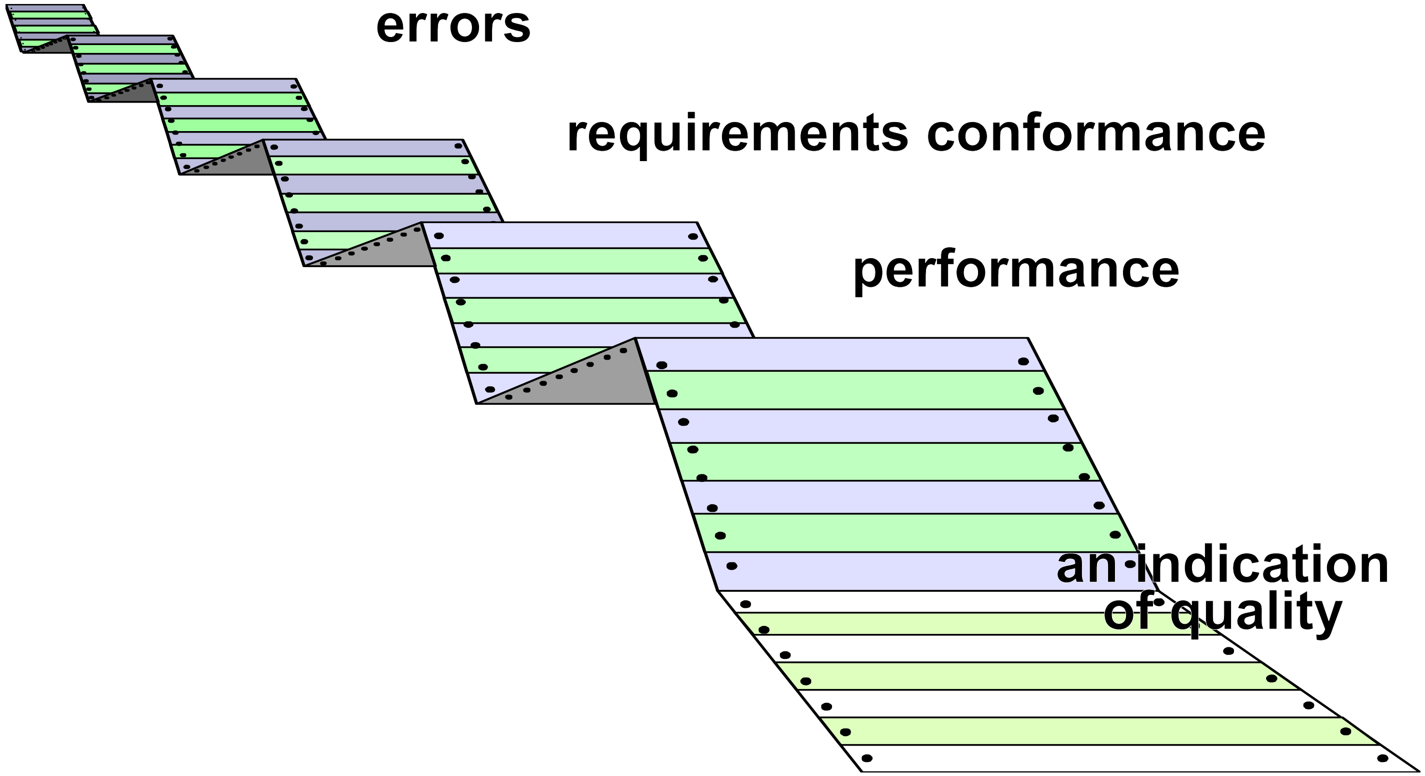
Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.

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.

What Testing Shows



errors

requirements conformance

performance

an indication of quality

V & V

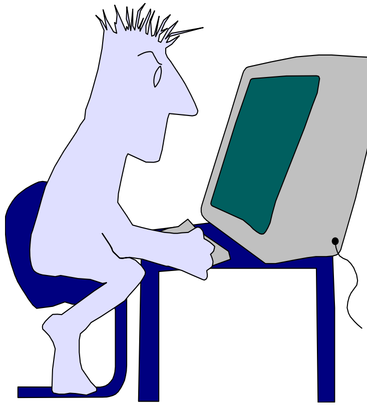
Software testing is part of a broader group of activities called verification and validation that are involved in software quality assurance

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

Understands the system

but, will test "gently"

and, is driven by "delivery"

independent tester

Must learn about the system,

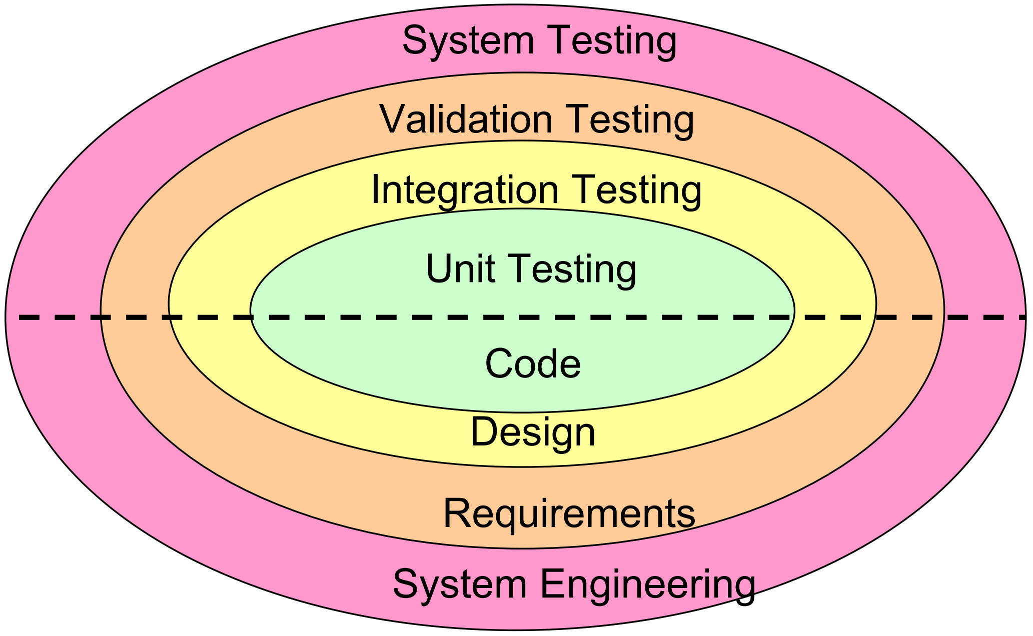
but, will attempt to break it

and, is driven by quality

Organizing for Software Testing

- Testing should aim at "breaking" the software
- Common misconceptions
  - The developer of software should do no testing at all
  - The software should be given to a secret team of testers who will test it unmercifully
  - The testers get involved with the project only when the testing steps are about to begin
- Reality: Independent test group
  - Removes the inherent problems associated with letting the builder test the software that has been built
  - Removes the conflict of interest that may otherwise be present
  - Works closely with the software developer during analysis and design to ensure that thorough testing occurs

Testing Strategy



Criteria for Completion of Testing

- When are we finished testing? : No definite answer for this question.
- But few responses:
  - You are never done testing; the burden simply shifts from developer/tester to end user.
  - You are done testing when you run out of time or money.
- Rigorous criteria: *Cleanroom software engineering approach.*
  - Statistical testing method, executes the series of tests derived from a statistical sample of all possible program executions by all users

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

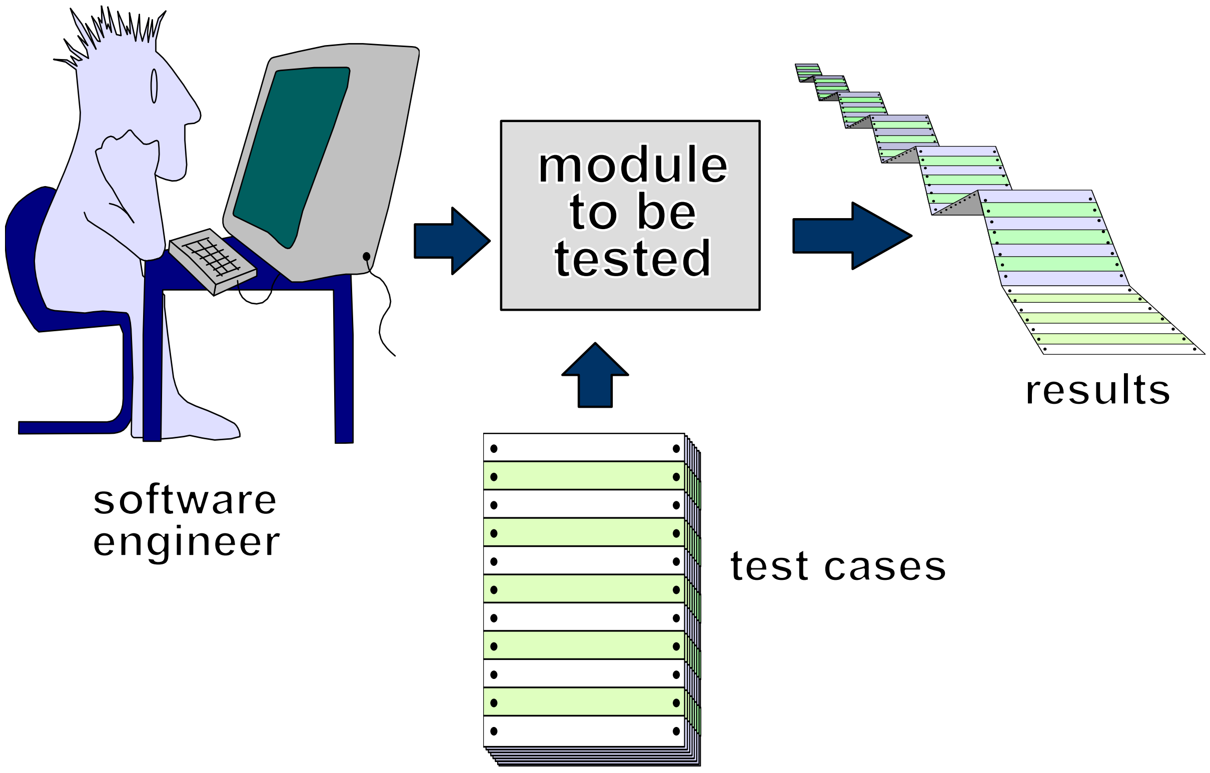
- Specify product requirements in a quantifiable manner long before testing commences.
- State testing objectives explicitly.
- Understand the users of the software and develop a profile for each user category.
- Develop a testing plan that emphasizes “rapid cycle testing.”
- Build “robust” software 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.

Test Strategies for Conventional Software

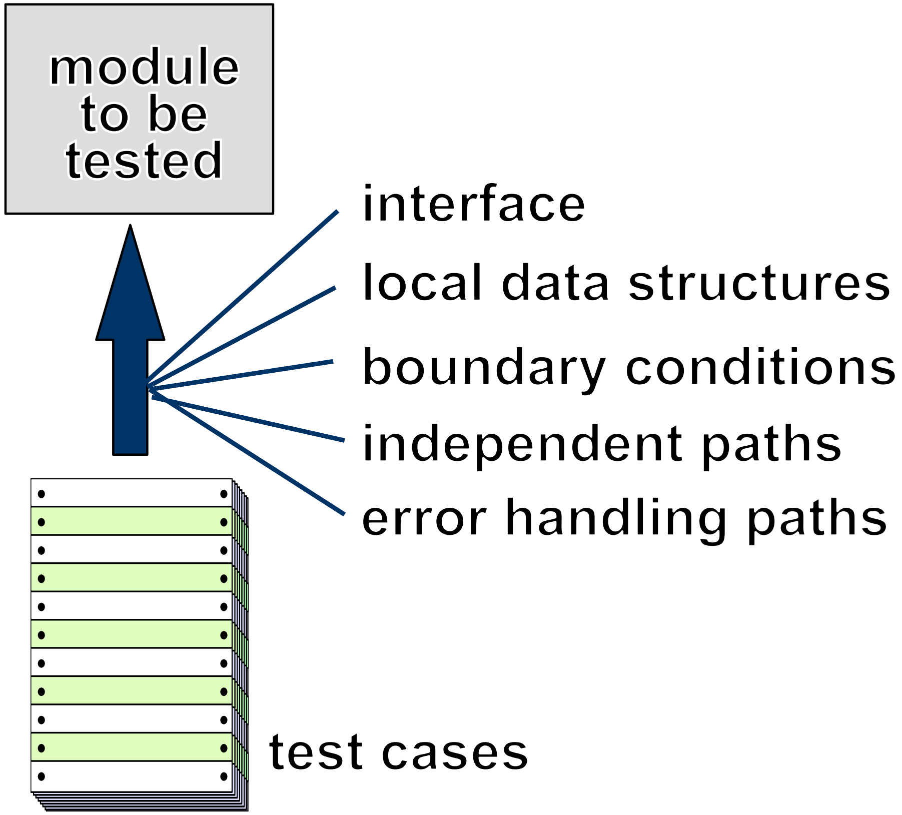
Unit Testing

- 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

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

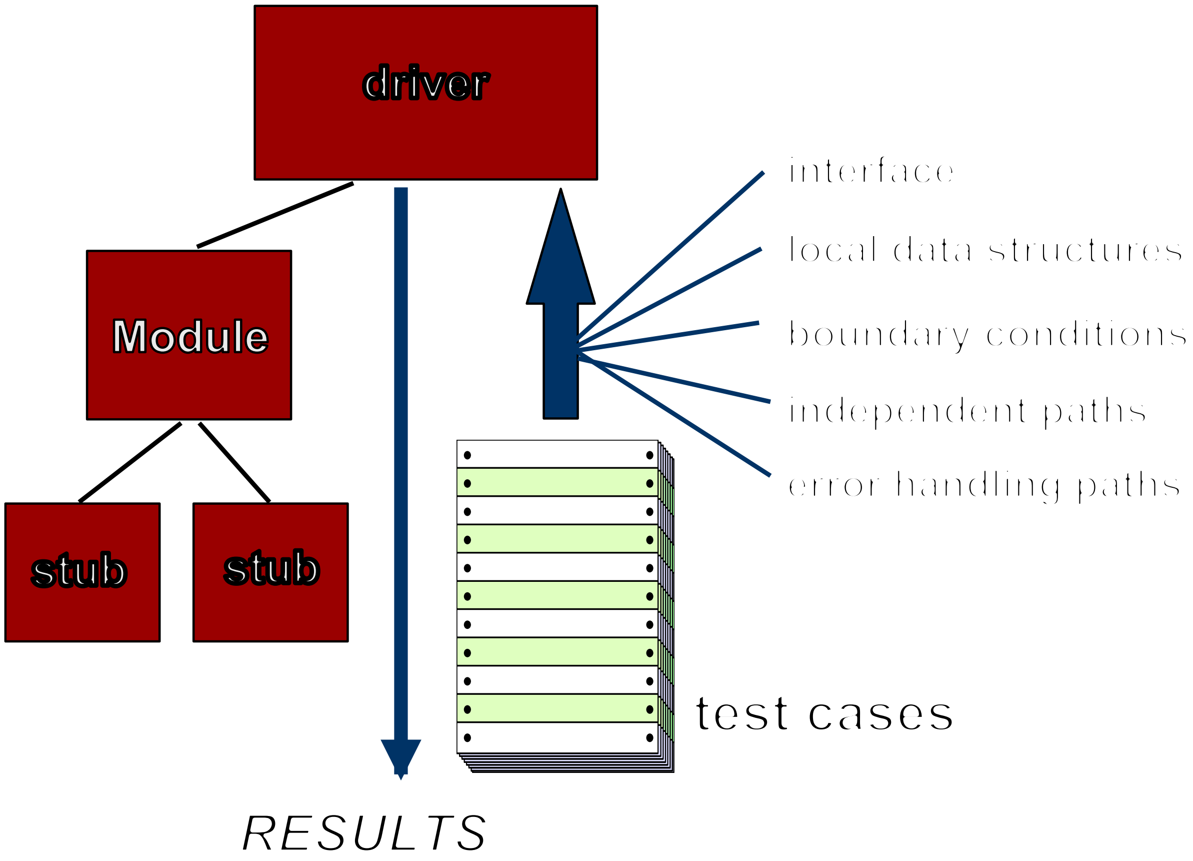
Exception condition processing is incorrect

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

Unit Test Environment



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

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

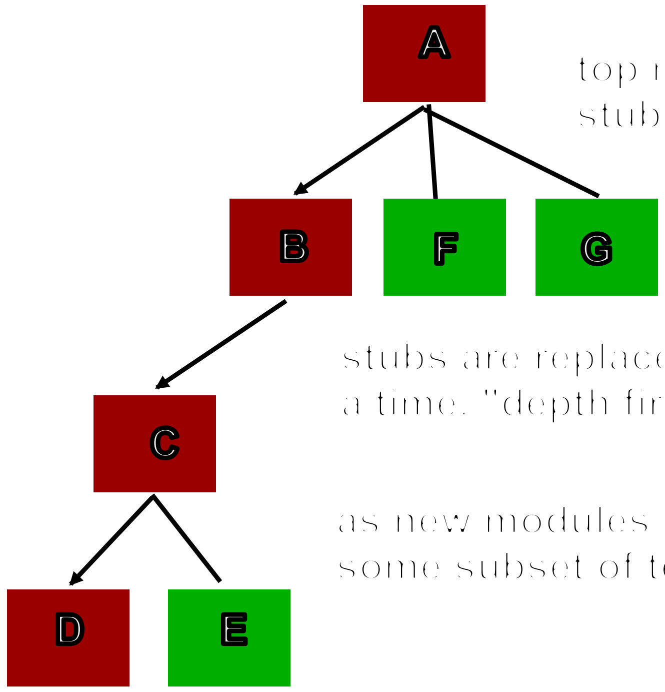
Incremental Integration Testing

- 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



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

Integration and testing starts with the most atomic modules in the control hierarchy

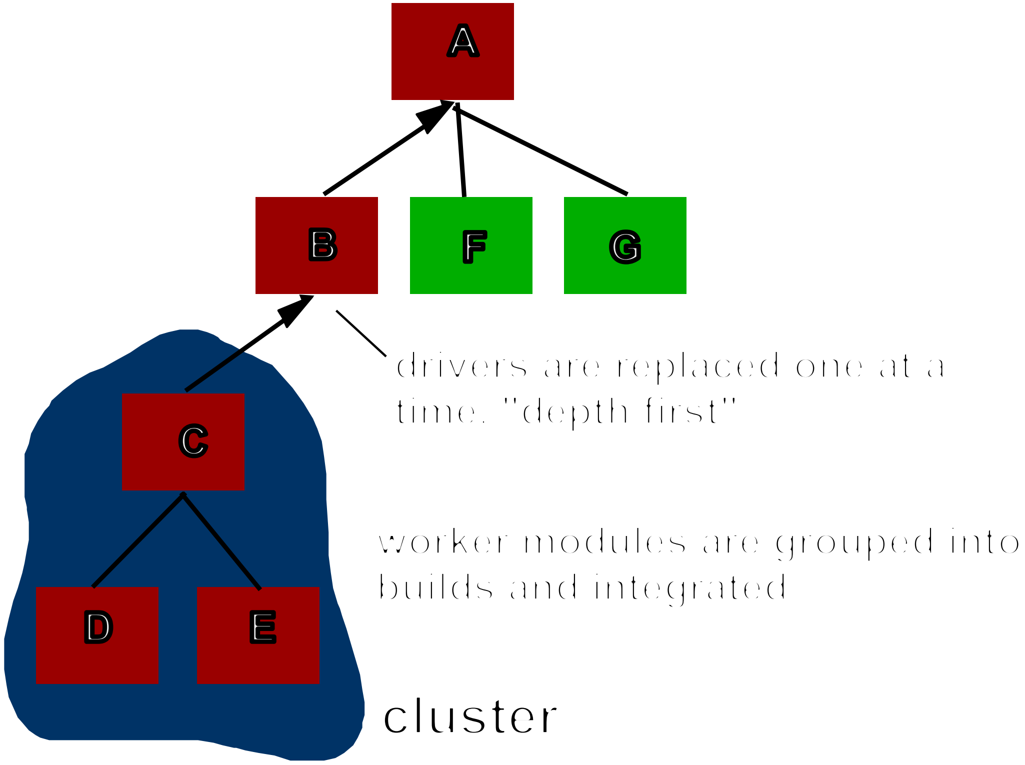
Advantages

- 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

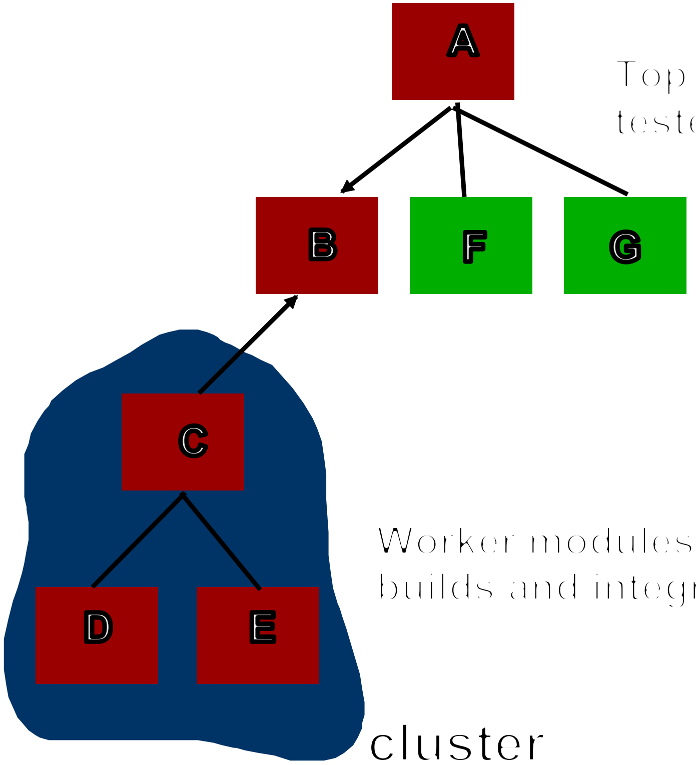
Bottom-Up Integration


A
B
F
G
C
D
E
drivers are replaced one at a time, "depth first"
worker modules are grouped into builds and integrated
cluster

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



Top modules are tested with stubs

Worker modules are grouped into builds and integrated

cluster

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

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

Test Strategies for Object-Oriented Software

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

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

System Testing

Different Types

- 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 reinitialization, checkpointing 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

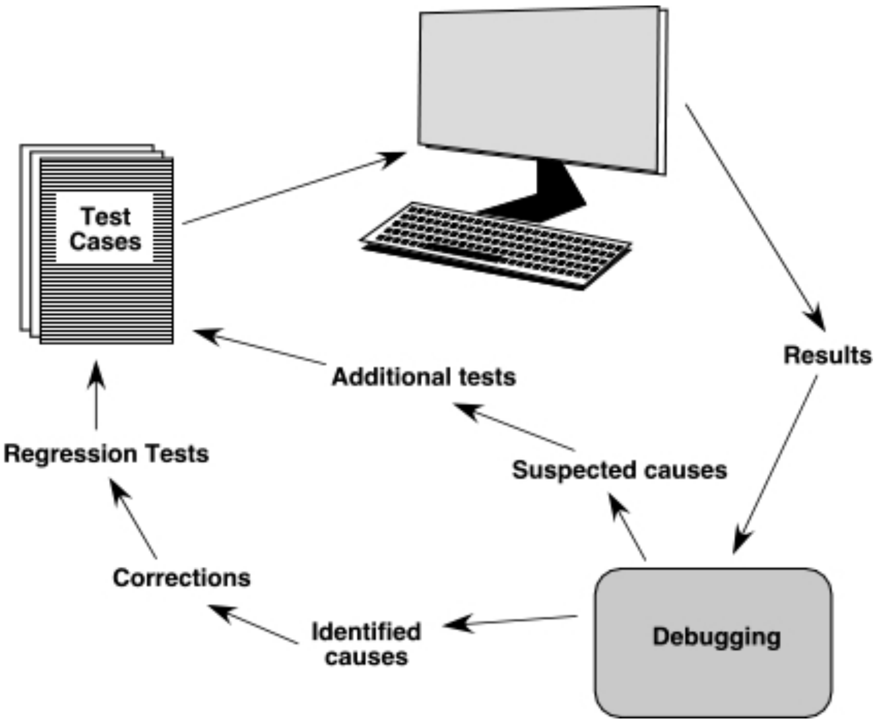
- Executes a system in a manner that demands resources in abnormal quantity, frequency, or volume

Performance testing

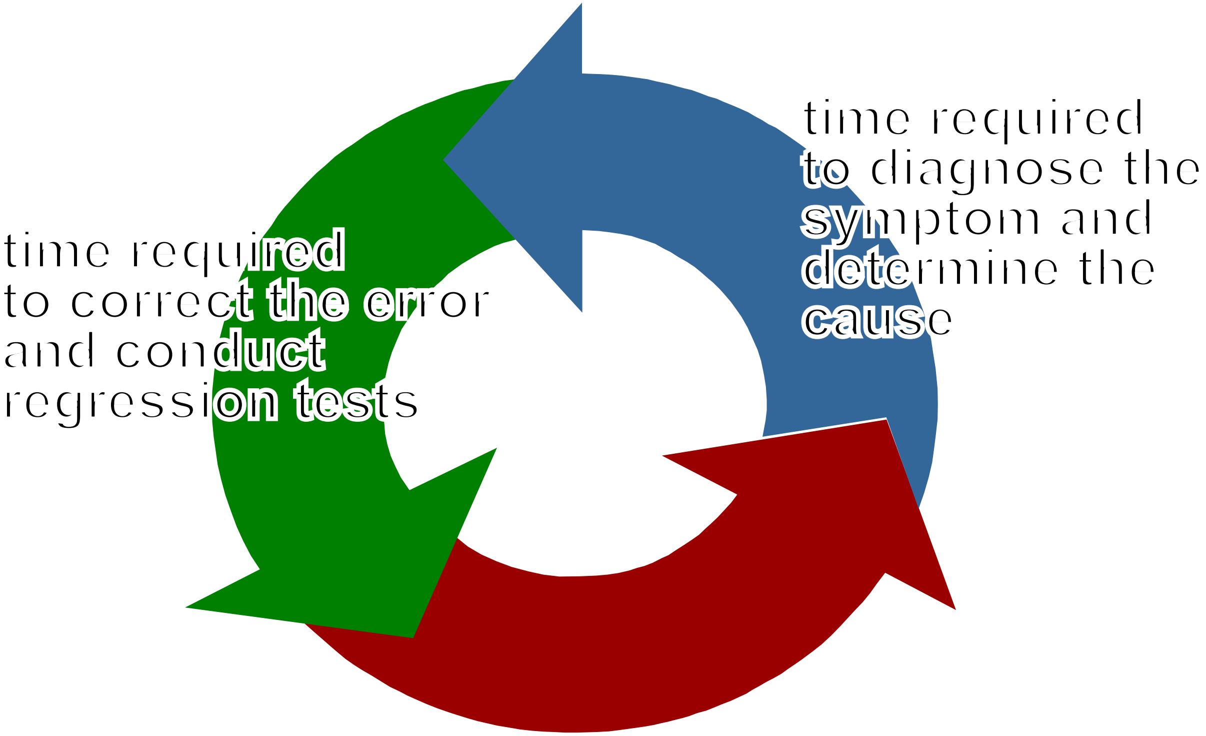
- Tests the run-time performance of software within the context of an integrated system
- Often coupled with stress testing and usually requires both hardware and software instrumentation
- Can uncover situations that lead to degradation and possible system failure

Debugging: A Diagnostic Process

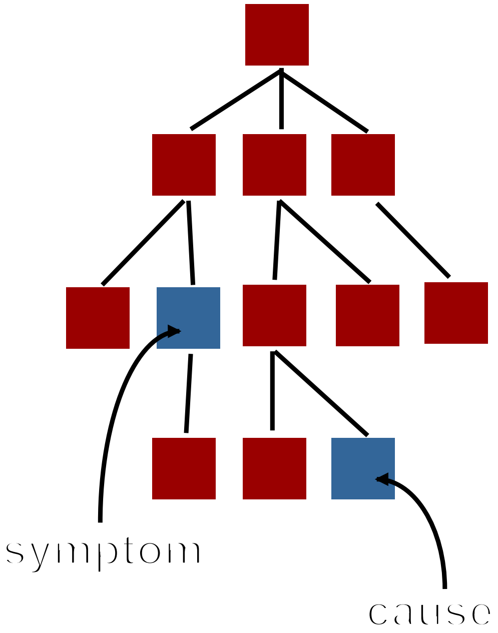
The Debugging Process



Debugging Effort

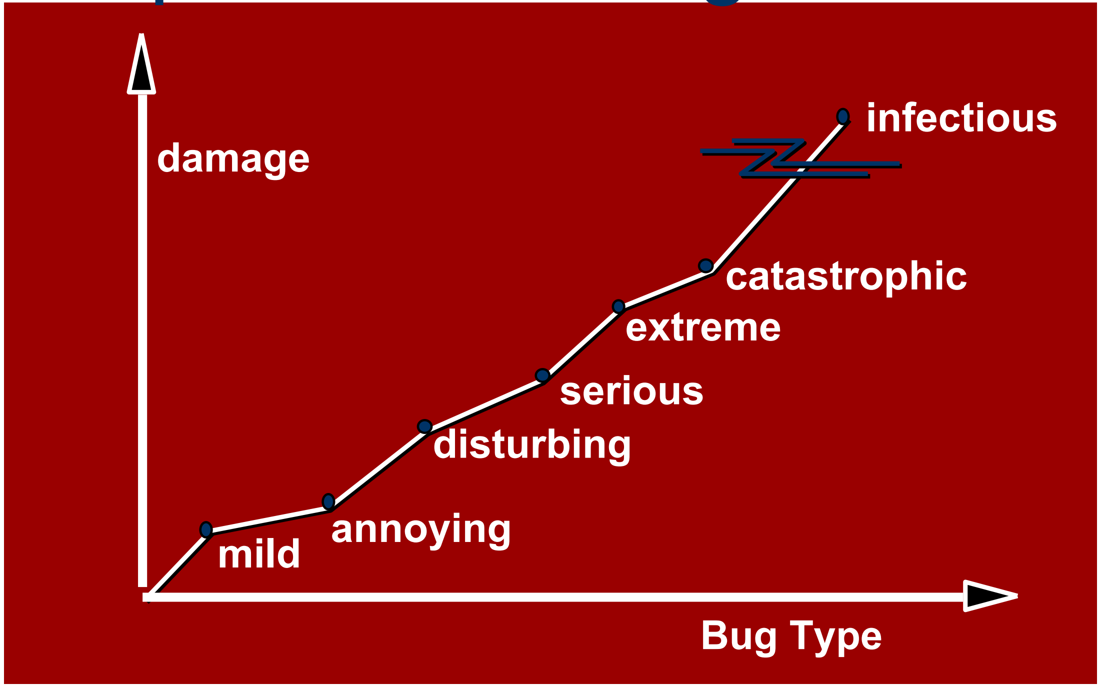

time required to correct the error and conduct regression tests
time required to diagnose the symptom and determine the cause

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: Let computer find the error
- Backtracking: Source code is traced backward until the cause or error will found
- Cause Elimination: A cause hypothesis
    - Induction or Deduction
    - Data is used to prove or disprove the hypothesis

Automated Debugging

Different tools

Integrated Development Environments (IDEs)

Debugging compilers

Dynamic Debugging aids

Automated test case generators

Cross reference mapping tools

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