Unit IV: Software Testing

CHAPTER 19: QUALITY CONCEPTS

19.1 What Is Quality?

19.2 Software Quality

19.2.2 McCall's Quality Factors

CHAPTER 22 SOFTWARE TESTING STRATEGIES

22.1 A Strategic Approach to Software Testing

22.1.1 Verification and Validation

22.1.2 Organizing for Software Testing

22.1.3 Software Testing Strategy—The Big Picture

22.1.4 Criteria for Completion of Testing

22.2 Strategic Issues

22.3 Test Strategies for Conventional Software

22.3.1 Unit Testing

22.3.2 Integration Testing

22.7 Validation Testing

22.7.1 Validation-Test Criteria

22.7.2 Configuration Review

22.7.3 Alpha and Beta Testing

22.8 System Testing

22.8.1 Recovery Testing

22.8.2 Security Testing

22.8.3 Stress Testing

22.8.4 Performance Testing

22.8.5 Deployment Testing

22.9 The Art of Debugging

22.9.1 The Debugging Process

22.9.2 Psychological Considerations

22.9.3 Debugging Strategies

22.9.4 Correcting the Error

CHAPTER 23 TESTING CONVENTIONAL APPLICATIONS

23.1 Software Testing Fundamentals
23.3 White-Box Testing
23.4 Basis Path Testing
- 23.4.1 Flow Graph Notation
- 23.4.2 Independent Program Paths
- 23.4.3 Deriving Test Cases
- 23.4.4 Graph Matrices

23.5 Control Structure Testing

Unit IV: Software Testing

CHAPTER 22 SOFTWARE TESTING STRATEGIES

22.1 A Strategic Approach to Software Testing
- 22.1.1 Verification and Validation
- 22.1.2 Organizing for Software Testing
- 22.1.3 Software Testing Strategy—The Big Picture
- 22.1.4 Criteria for Completion of Testing

22.2 Strategic Issues

22.3 Test Strategies for Conventional Software
- 22.3.1 Unit Testing
- 22.3.2 Integration Testing

22.7 Validation Testing
- 22.7.1 Validation-Test Criteria
- 22.7.2 Configuration Review
- 22.7.3 Alpha and Beta Testing

22.8 System Testing
- 22.8.1 Recovery Testing
- 22.8.2 Security Testing
- 22.8.3 Stress Testing
- 22.8.4 Performance Testing
- 22.8.5 Deployment Testing

22.9 The Art of Debugging
- 22.9.1 The Debugging Process
- 22.9.2 Psychological Considerations
- 22.9.3 Debugging Strategies
- 22.9.4 Correcting the Error

Preamble

- "approaches and philosophies" are what we call *strategy*

Software Testing Definition

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

Software testing is the process of executing a software system to determine whether it matches its specification and executes in its intended environment.

22.1 A Strategic Approach To Software Testing

- 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.

22.1 A Strategic Approach To Software Testing

22.1.1 Verification and Validation

- Software testing is one element of a broader topic that is often referred to as verification and validation (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?"

22.1 A Strategic Approach To Software Testing

22.1.2 Organizing for Software Testing (Quality Assurance Organization)

developer

Understands the system

but, will test "gently"

and, is driven by "delivery"

independent tester

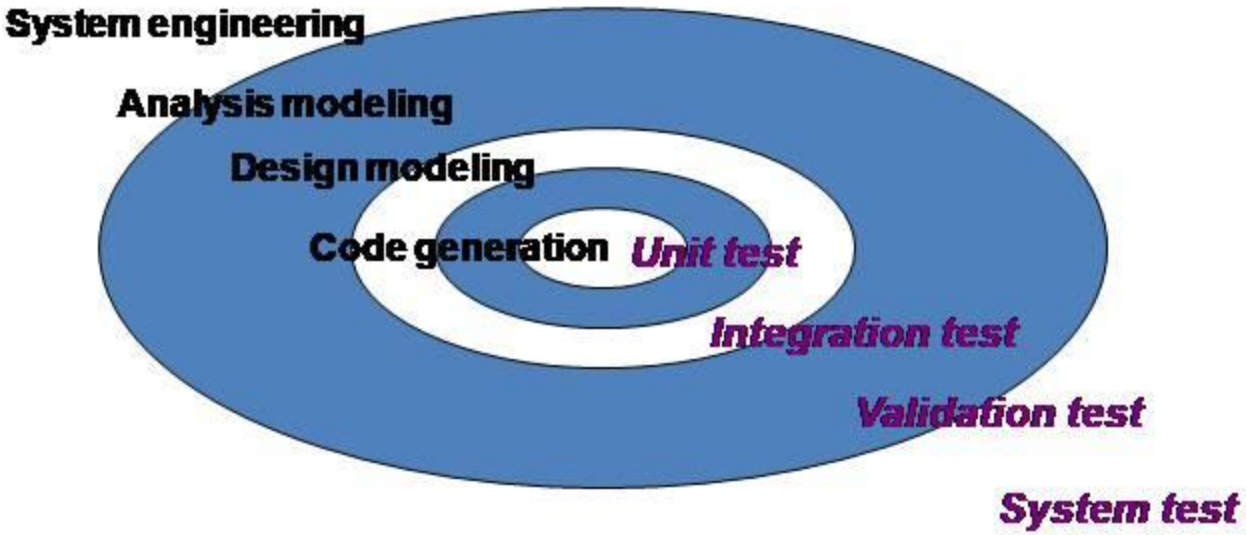
Must learn about the system,

but, will attempt to break it

and, is driven by quality

22.1 A Strategic Approach To Software Testing

22.1.3 Software Testing Strategy—The Big Picture



22.1 A Strategic Approach To Software Testing

22.1.3 Software Testing Strategy—The Big Picture

- **A strategy** for software testing may also be viewed in the context of the spiral.
  - ***Unit testing begins at the vortex*** *of the spiral and concentrates on* each unit (e.g., component, class, or WebApp content object).
  - Testing progresses by moving outward along the spiral to ***integration testing,*** *where the focus is on design and the construction of* the software architecture.
  - Then ***validation testing,*** *where requirements established as part of requirements* modeling are validated against the software that has been constructed.
  - Finally, ***system testing,*** *where the software and other system elements* are tested as a whole.

22.1.3 Software Testing Strategy—The Big Picture

- 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

22.1.4 Criteria for Completion of Testing

- A classic question arises every time software testing is discussed: "When are we done testing—how do we know that we've tested enough?" Sadly, there is **no definitive answer** to this question, but there are a few pragmatic responses and early attempts at empirical guidance.

22.2 Strategic Issues

- Specify product requirements in a quantifiable manner long before testing commences.
- State testing objectives explicitly.

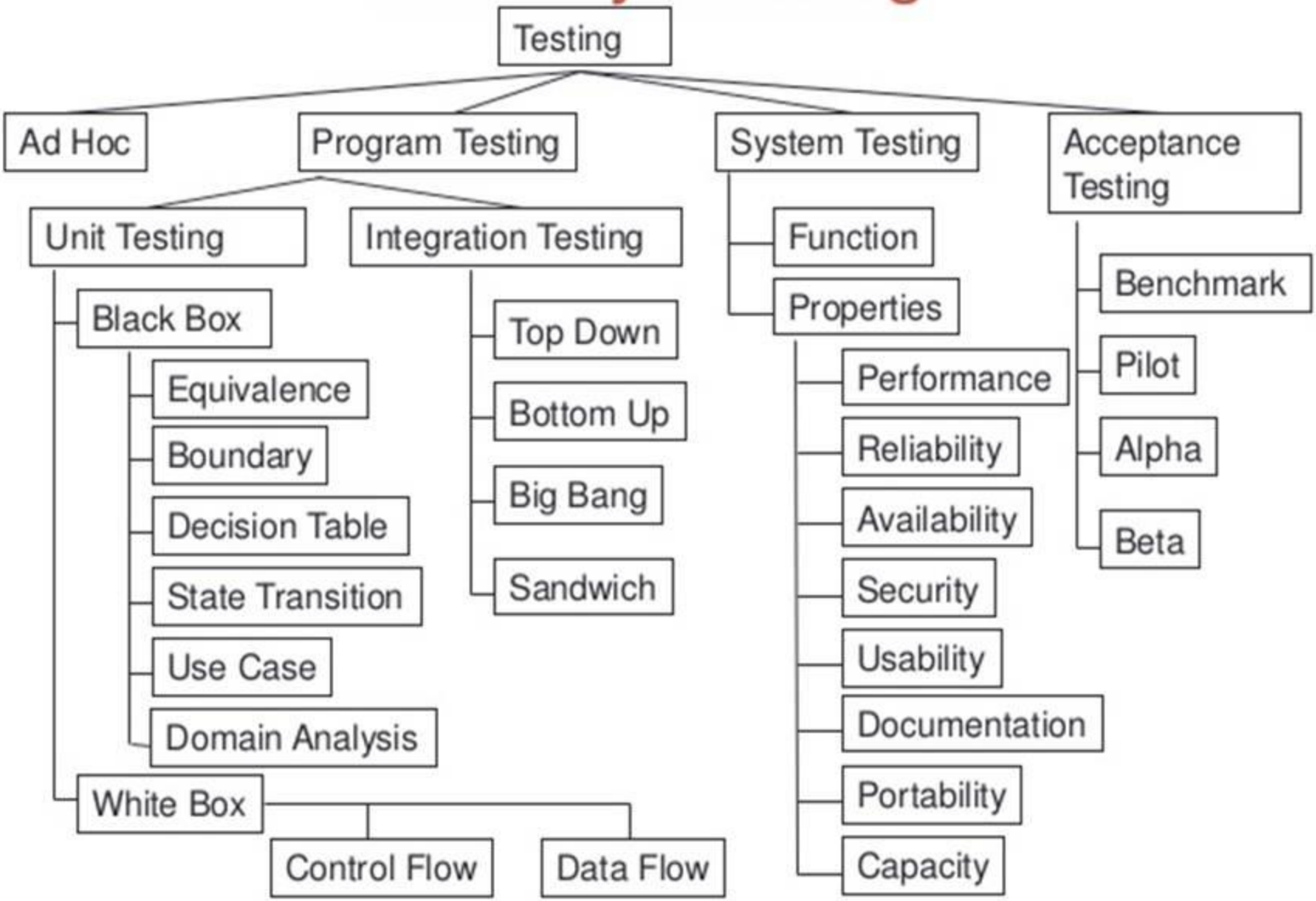
- Understand the users of the software and develop a profile for each user category.
- Develop a testing plan that emphasizes "rapid cycle testing."

- Build "robust" software that is designed to test itself.
- Use effective technical reviews as a filter prior to testing.

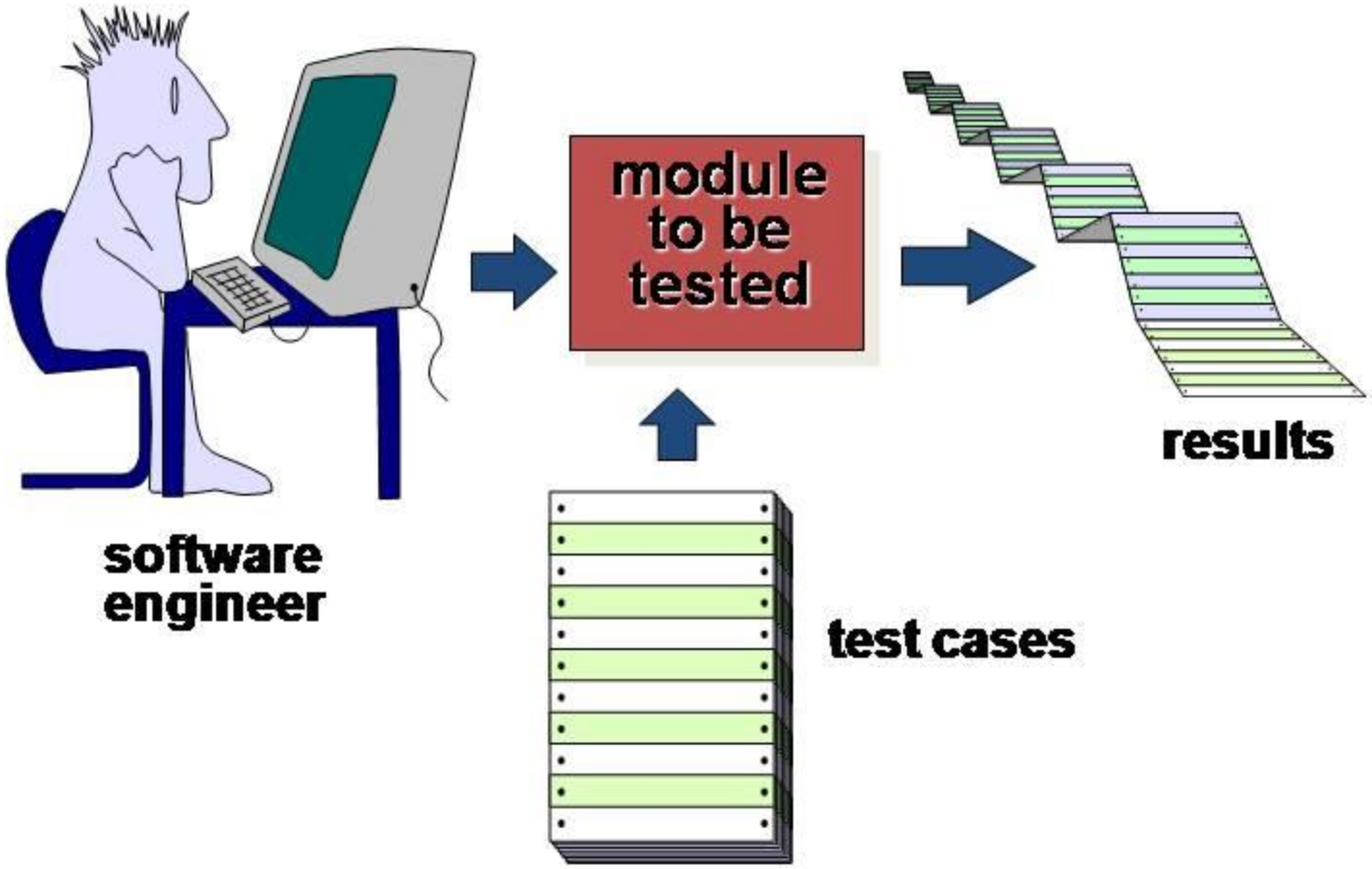
- Conduct technical reviews to assess the test strategy and test cases themselves.
- Develop a continuous improvement approach for the testing process.

22.3 Test Strategies for Conventional Software

Hierarchy of Testing



a) Unit Testing

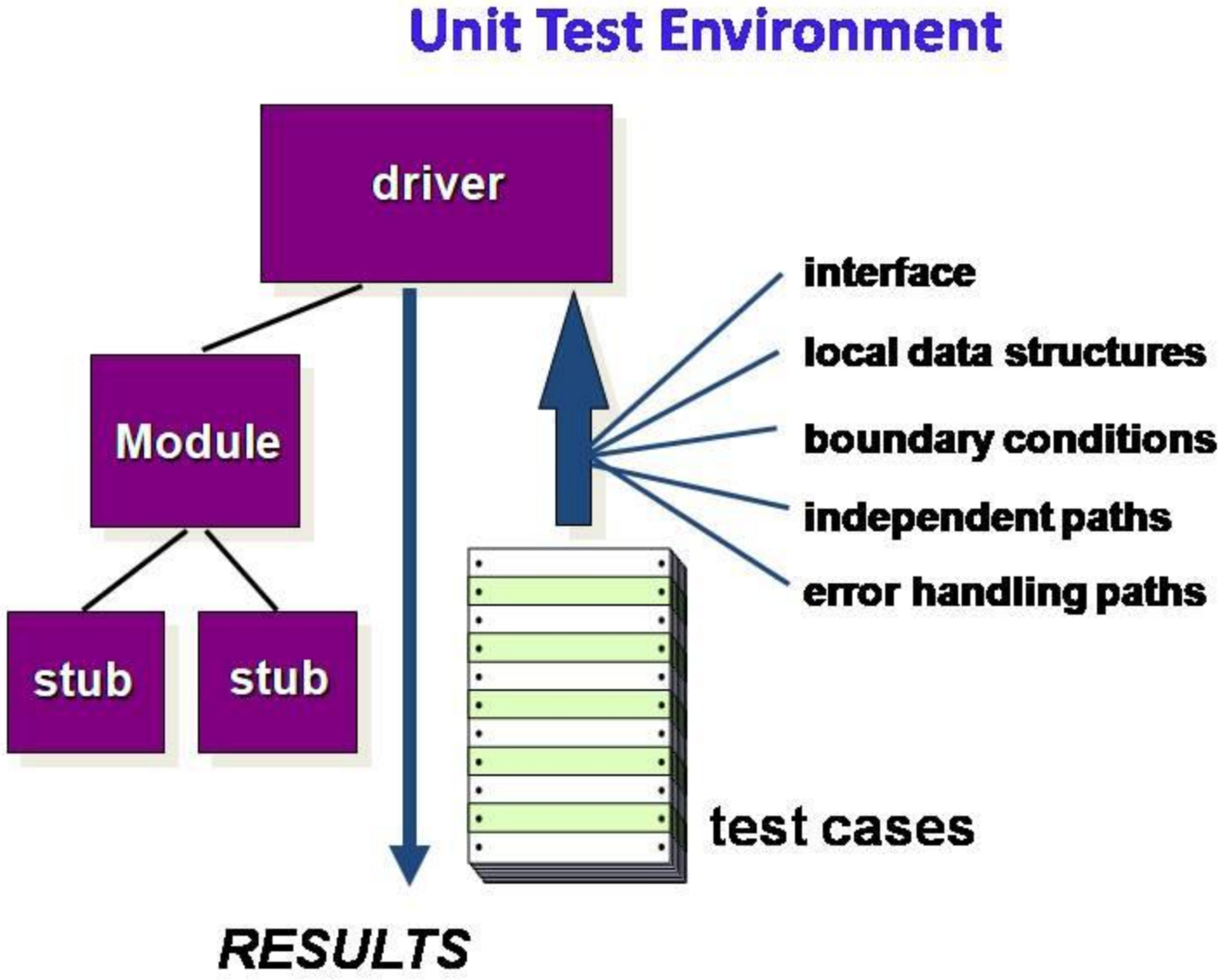


22.3 Test Strategies for Conventional Software

a) Unit Testing

- Focuses on each component "unit" individually.
- Heavily depends on white-box testing.
  - Tests of all components can be conducted in parallel.
  - Use the component level design description, found in the design document, as a guide to testing.

a) Unit Testing



22.3 Test Strategies for Conventional Software

a) Unit Testing

Unit-Test Procedure

- **Interface to the module - does information flow in and out properly?**
- **Local data structures - do local structures maintain data correctly during the algorithms execution?**
- **Boundary conditions - all boundary conditions should be tested (loops, First and last record, array limits).**
- **Independent paths- Try to execute each statement at least once. Look at all the paths through the module.**
- **Error Handling paths- Trigger all possible errors and see that they are handled properly,**
  - messages are helpful and correct,
  - exception-condition processing is correct
  - error messages identify the location of the true error.

22.3 Test Strategies for Conventional Software

a) Unit Testing

Unit-Test Procedure

- **Stubs and Drivers:**
  - Additional SW used to help test components.
- **Driver**
  - a main program that passes data to the component and displays or prints the results.
- **Stub**
  - a "dummy" sub-component or sub-program used to replace components/programs that are subordinate to the unit being tested.
  - The stub may do minimal data manipulation, print or display something when it is called and then return control to the unit being tested.

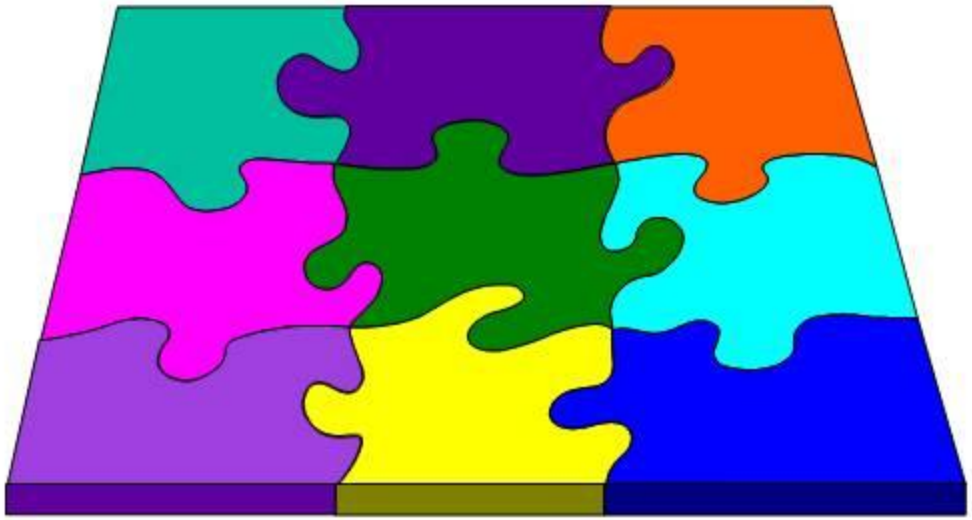
b) Integration Testing

- Systematic approach for constructing program structure while conducting tests to uncover errors associated with interfacing.
- Testing focuses on the design and construction of the SW architecture.
  - This is when we fit the units together.
- Integration testing includes both
  - verification and
  - program construction.

b) Integration Testing

Options:

- **the "big bang" approach**
- **an incremental construction strategy**

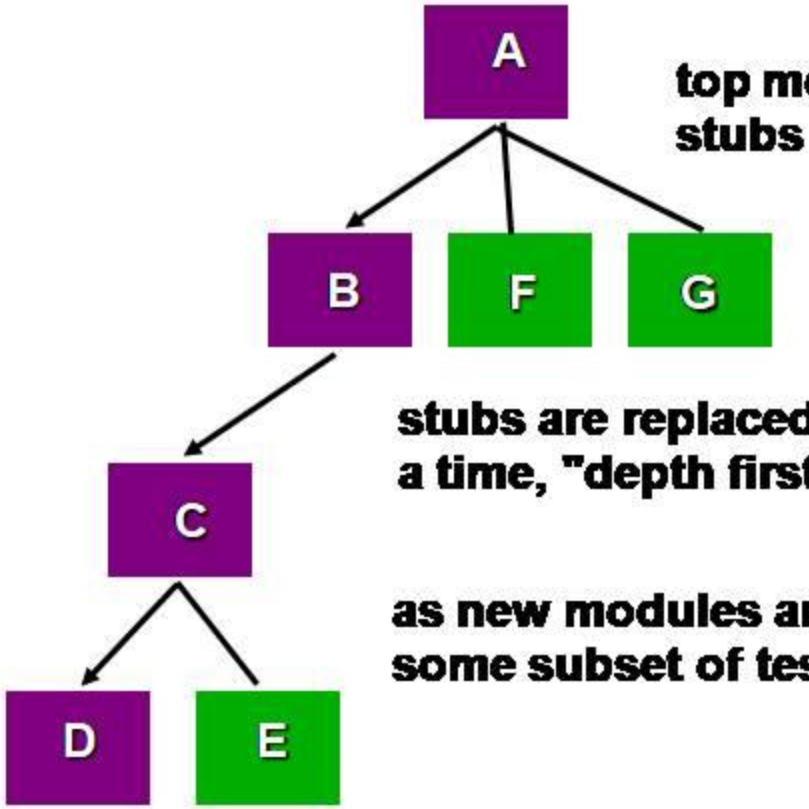


b) Integration Testing

- Black-box testing techniques are mostly used.
  - Some white-box testing may be used for major control paths.
  - Look to the SW design specification for information to build test cases.
- *Incremental* integration - program is constructed and tested in small segments.
  - Top-Down Integration testing
  - Bottom-Up Integration testing

b) Integration Testing

Top Down Integration



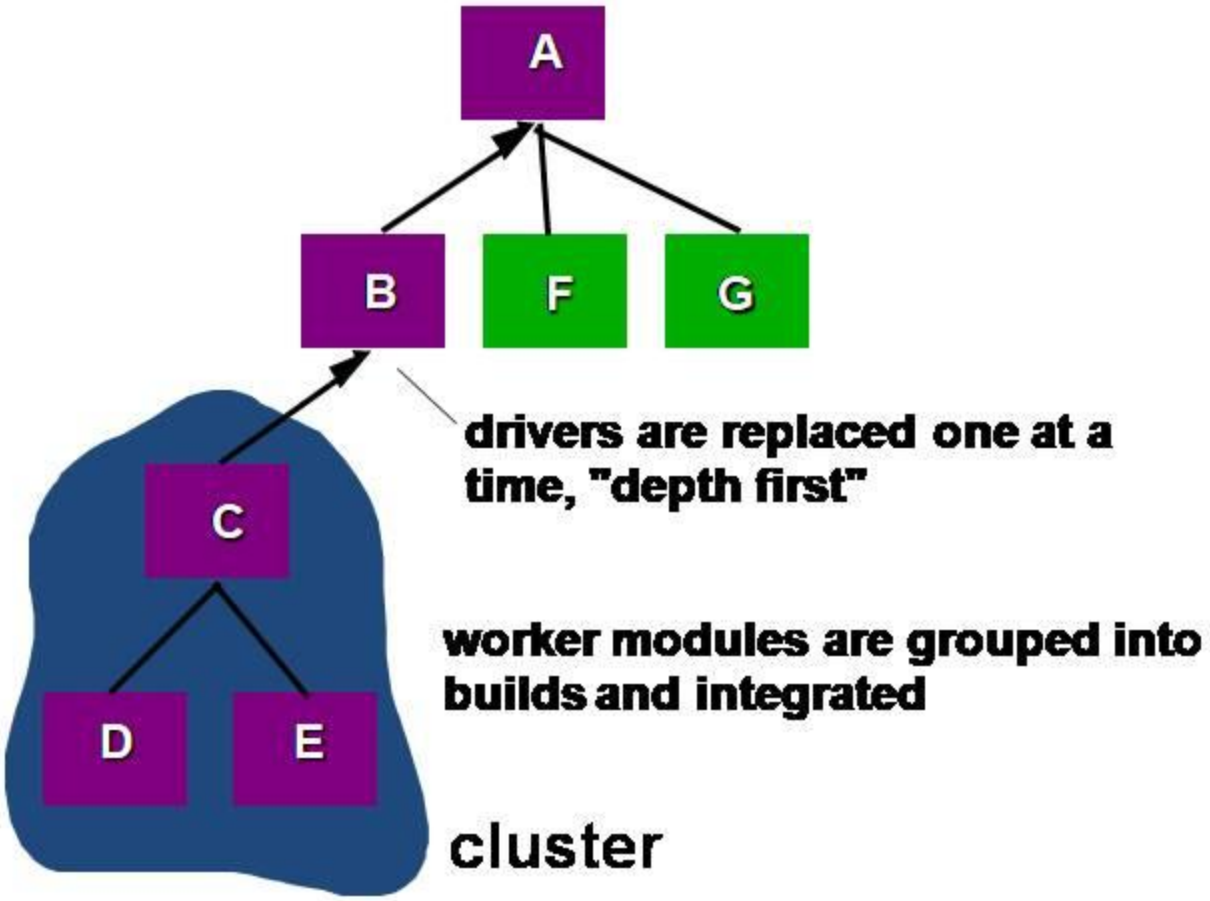
top module is tested with stubs

stubs are replaced one at a time, "depth first" (or breadth first)

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

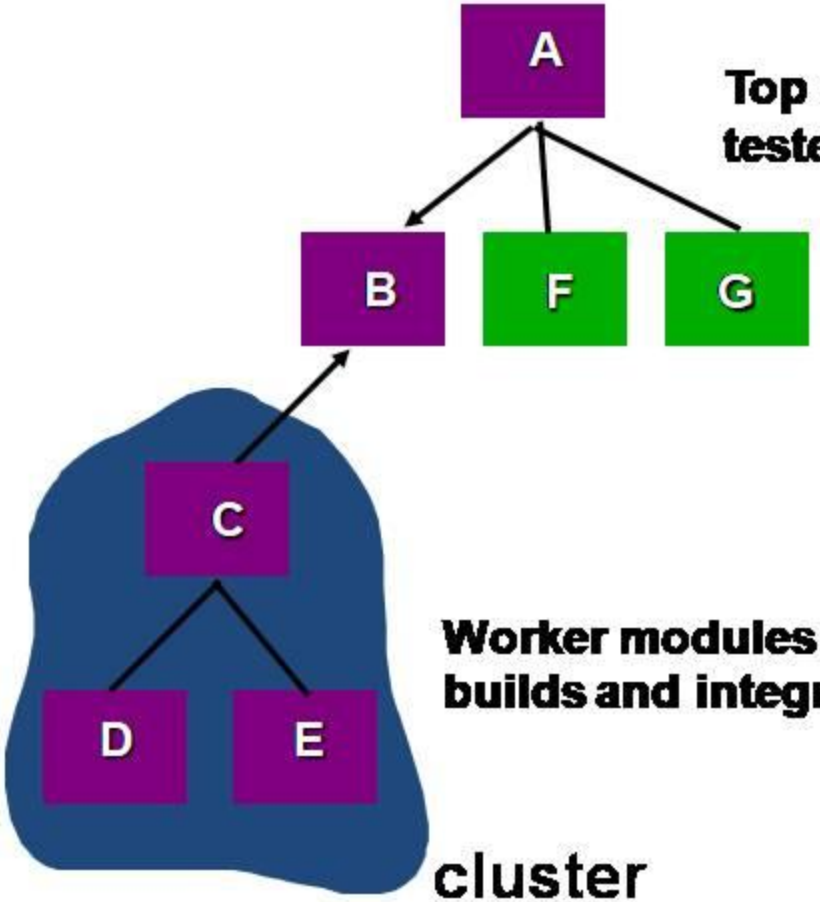
b) Integration Testing

Bottom-Up Integration



b) Integration Testing

Bottom-Up Integration



Top modules are tested with stubs

Worker modules are grouped into builds and integrated

c) Regression Testing

- 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.

- It 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.

d) 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.

22.7 Validation Testing

- Ensures that all functional, behavioural and performance requirements of the system were satisfied.
  - These were detailed in the SW requirements specification.
  - Look in the validation criteria section
- Only Black-box testing techniques are used.
- Goal is to prove conformity with the requirements.

22.7 Validation Testing

- For each test case, one of two possibilities conditions will exist:
  - 1. Test for the function or performance criteria will conform to the specification and is accepted.
  - 2. A deviation will exist and a deficiency is uncovered.
    - This could be an error in the SW or a deviation from the specification
    - SW works but it does not do what was expected.

22.7 Validation Testing

- Alpha and Beta Testing
  - A series of acceptance tests to enable the customer to validate all requirements.
  - Conducted by the end users and not the developer/tester/system engineer.
- Alpha
  - At the developer site by customer
- Beta
  - At the customer site by the end user.
  - Live testing

22.8 System Testing

- Verifies that the new SW system integrates with the existing environment.
- This may include other systems, hardware, people and databases.
- Black-box testing techniques are used.
- Put your software in action with the reset of the system.
- Many types of errors could be detected but who will accept responsibility?

22.8 System Testing

System Testing Types

- **Recovery Testing - force the SW to fail in a** number of ways and verify that it recovers properly and in the appropriate amount of time.
- **Security Testing - attempt to break the security** of the system.
- **Stress Testing- execute the system in a way that** requires an abnormal demand on the quantity, frequency or volume of resources.
- **Performance Testing-** For real-time and embedded systems, test the run-time performance of the system.
- **Deployment Testing-** must execute on a variety of platforms. Also *called configuration testing, exercises the software in each environment in which it is to* operate.

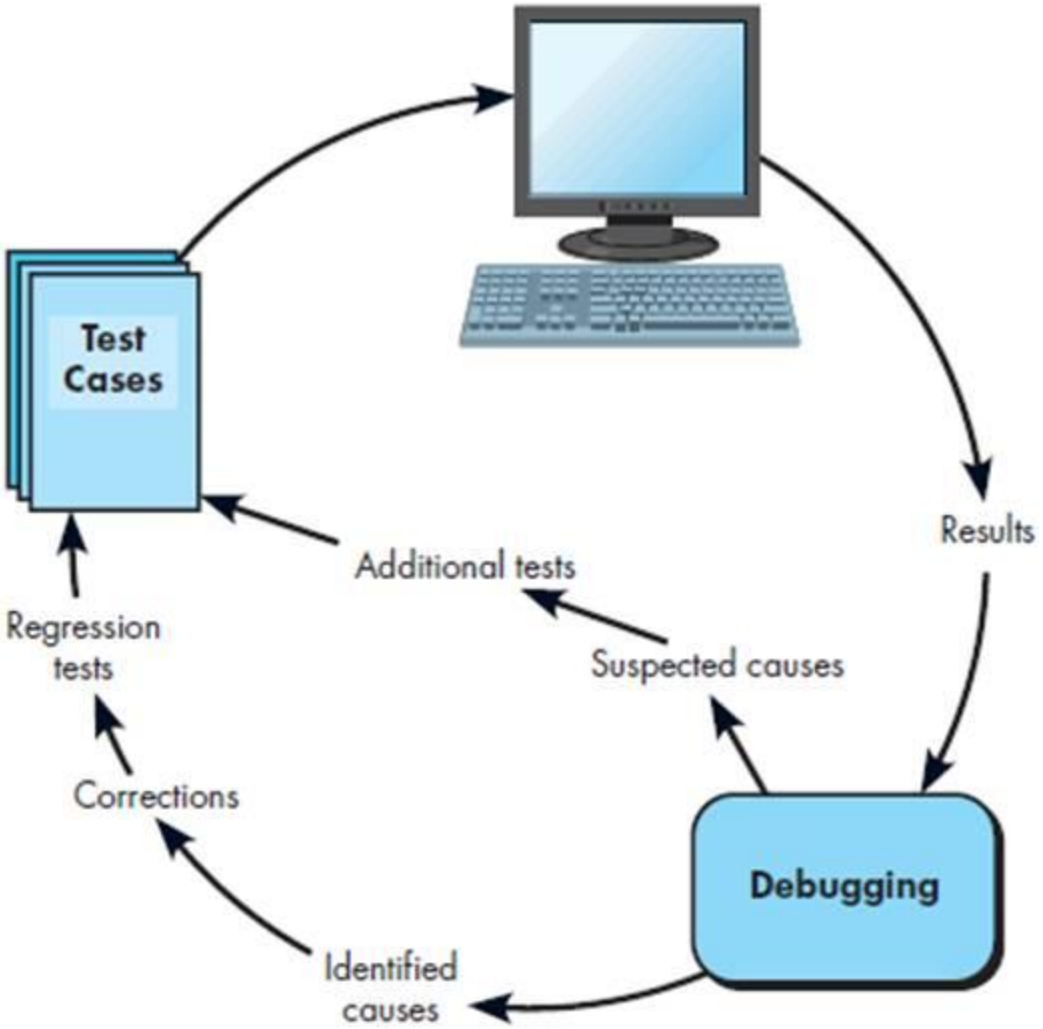
22.9 The Art of Debugging

- Debugging is a consequence of successful testing -- when a test case uncovers an error, it is the debugging process that results in the removal of the error.
- Debugging is an ART. The external manifestation of the error and the cause of the error normally do not share an obvious relationships.

The Debugging Process

FIGURE 22.7

The debugging process



22.9 The Art of Debugging

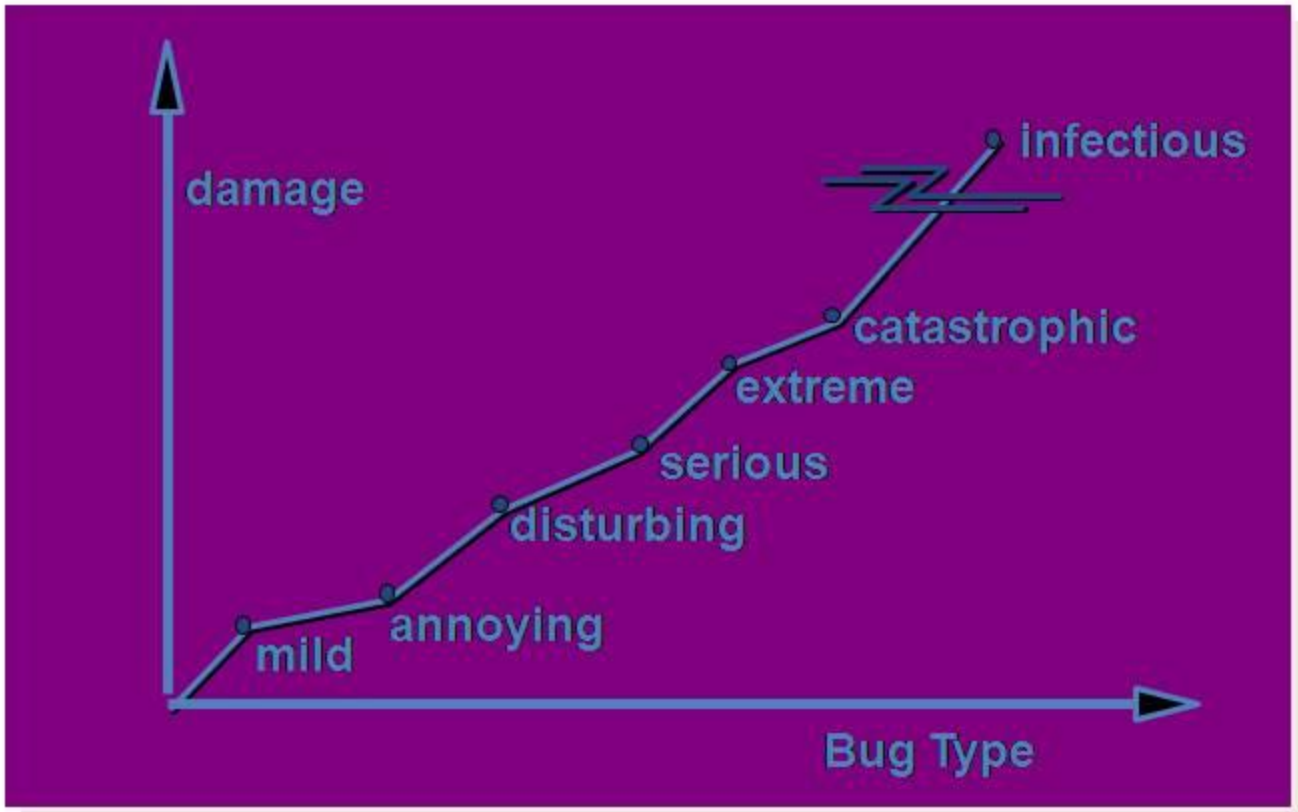
Characteristics of bugs :

1. Symptom and the cause may be geographically remote. That is, the symptom may appear in one part of a program, while the cause may actually be located at a site that is far removed. Highly coupled components exacerbate this situation.
2. Symptom may disappear (temporarily) when another error is corrected.
3. Symptom may actually be caused by nonerrors (e.g., round-off inaccuracies).
4. Symptom may be caused by human error that is not easily traced.

Characteristics of bugs :

5. Symptom may be a result of timing problems, rather than processing problems.
6. May be difficult to accurately reproduce input conditions (e.g., a real-time application in which input ordering is indeterminate).
7. Symptom may be intermittent (irregular). This is particularly common in embedded systems that couple hardware and software inextricably.
8. Symptom may be due to causes that are distributed across a number of tasks running on different processors.

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