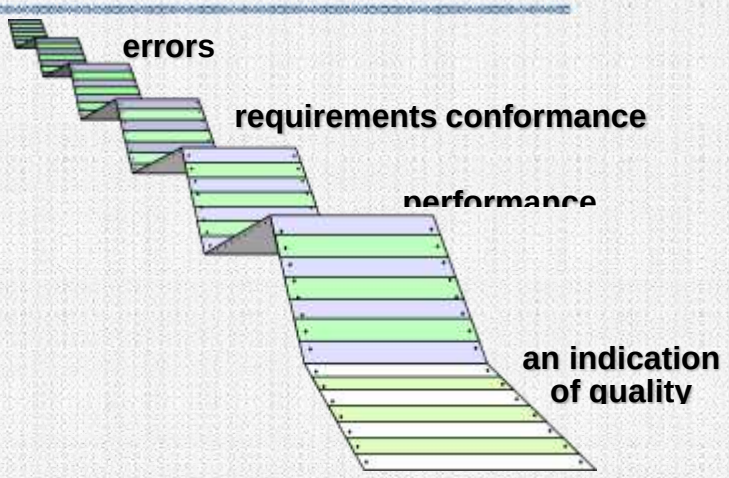
A strategic Approach to software testing

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

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

FIGURE 17.1
Testing strategy

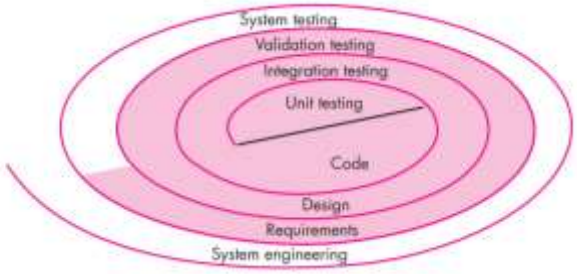
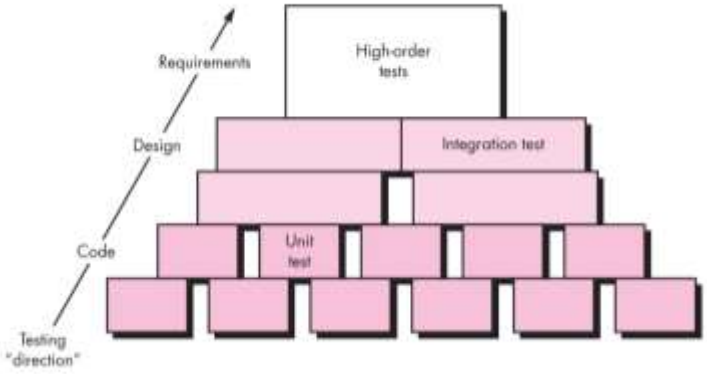


FIGURE 17.2
Software testing steps



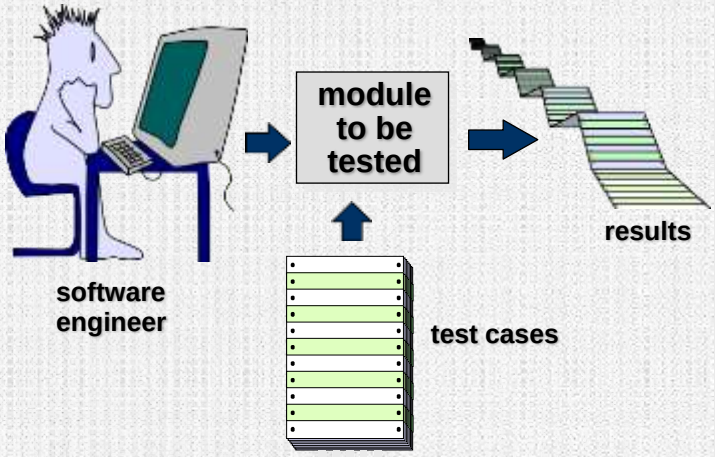
Testing Strategy

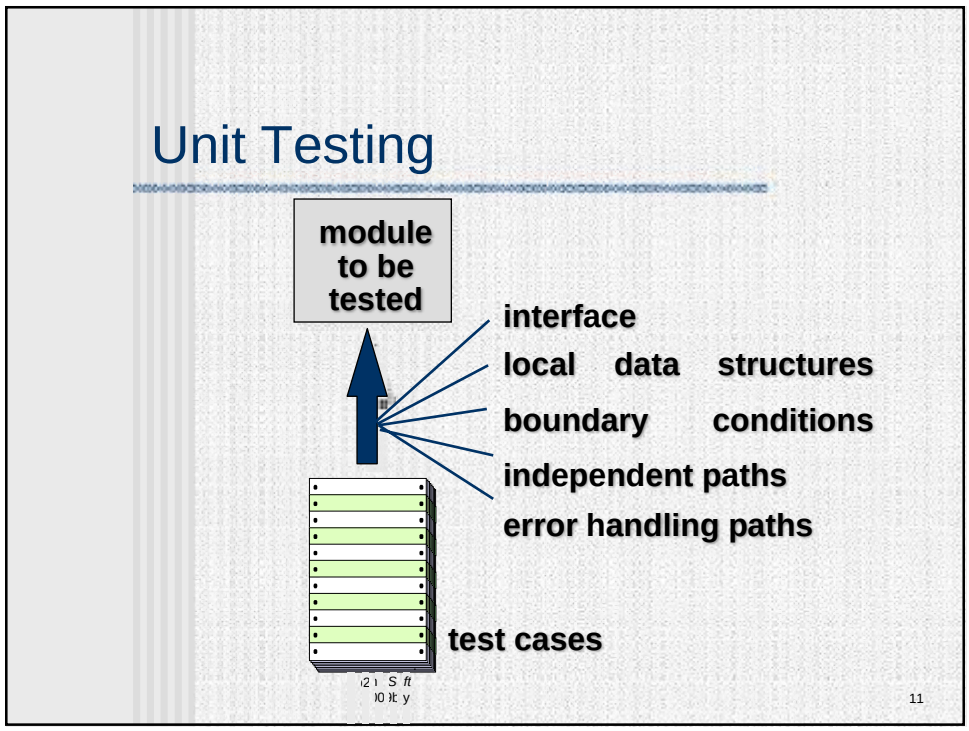
- We begin by 'testing-in-the-small' and move toward 'testing-in-the-large'
- For conventional software
  - The module (component) is our initial focus
  - Integration of modules follows
- For OO software
  - our focus when "testing in the small" changes from an individual module (the conventional view) to an OO class that encompasses attributes and operations and implies communication and collaboration

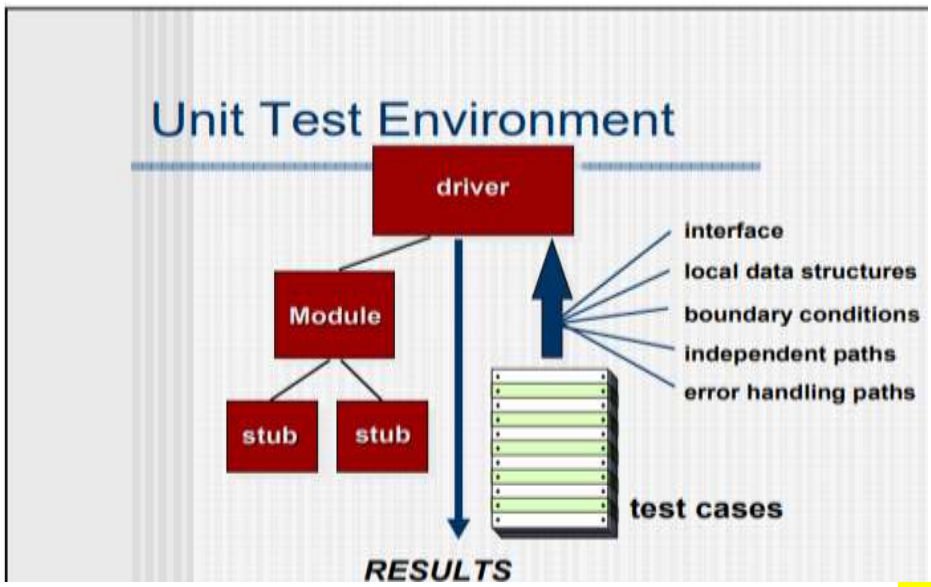
Strategic Issues

- 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







Stub: It refers to the section of the code that performs the imitation of the *called* function.

Driver: It refers to the section of the code that performs the imitation of the *calling* function.

Unit is the smallest testable part of the software system. Unit testing is done to verify that the lowest independent entities in any software are working fine. The smallest testable part is isolated from the remainder code and tested to determine whether it works correctly.

Why is Unit Testing important

Suppose you have two units and you do not want to test the units individually but as an integrated system to save your time.

Once the system is integrated and you found error in an integrated system it becomes difficult to differentiate that the error occurred in which unit so unit testing is mandatory before integrating the units.

When developer is coding the software it may happen that the dependent modules are not completed for testing,in such cases developers use stubs and drivers to simulate the called(stub) and caller(driver) units. Unit testing requires stubs and drivers, stubs simulates the called unit and driver simulates the calling unit.

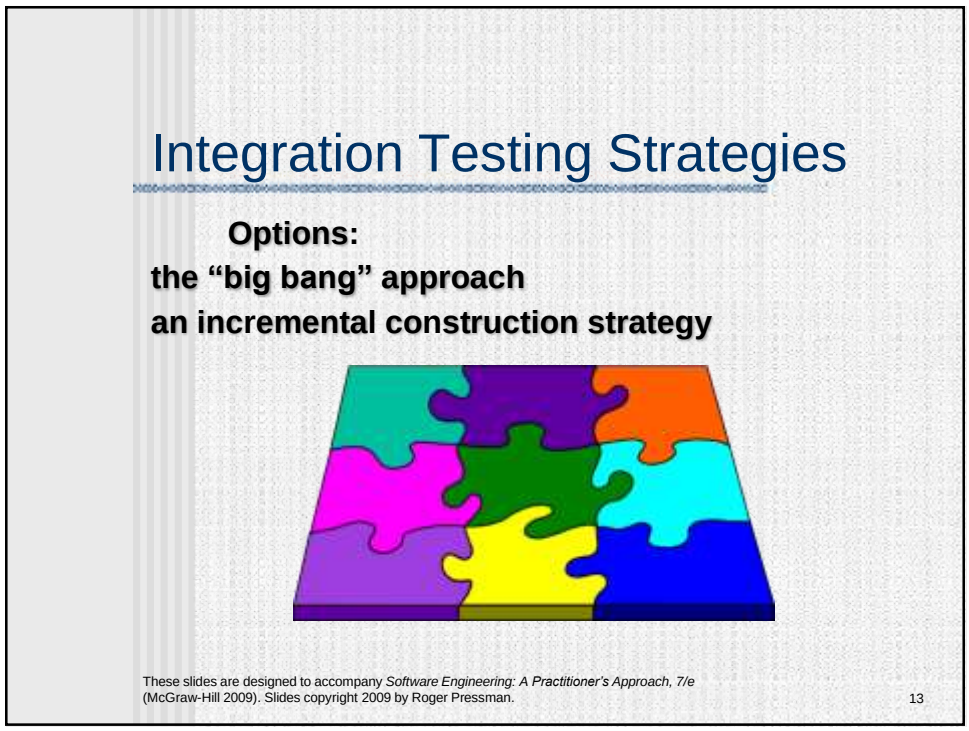
Lets explain STUBS and DRIVERS in detail.

STUBS:

Assume you have 3 modules, Module A, Module B and module C. Module A is ready and we need to test it, but module A calls functions from Module B and C which are not ready, so developer will write a dummy module which simulates B and C and returns values to module A. This dummy module code is known as stub.

DRIVERS:

Now suppose you have modules B and C ready but module A which calls functions from module B and C is not ready so developer will write a dummy piece of code for module A which will return values to module B and C. This dummy piece of code is known as driver.



What is Incremental Testing?

After unit testing is completed, developer performs integration testing. It is the process of verifying the interfaces and interaction between modules. While integrating, there are lots of techniques used by developers and one of them is the incremental approach.

In Incremental integration testing, the developers integrate the modules one by one using stubs or drivers to uncover the defects. This approach is known as incremental integration testing. To the contrary, big bang is one other integration testing technique, where all the modules are integrated in one shot.

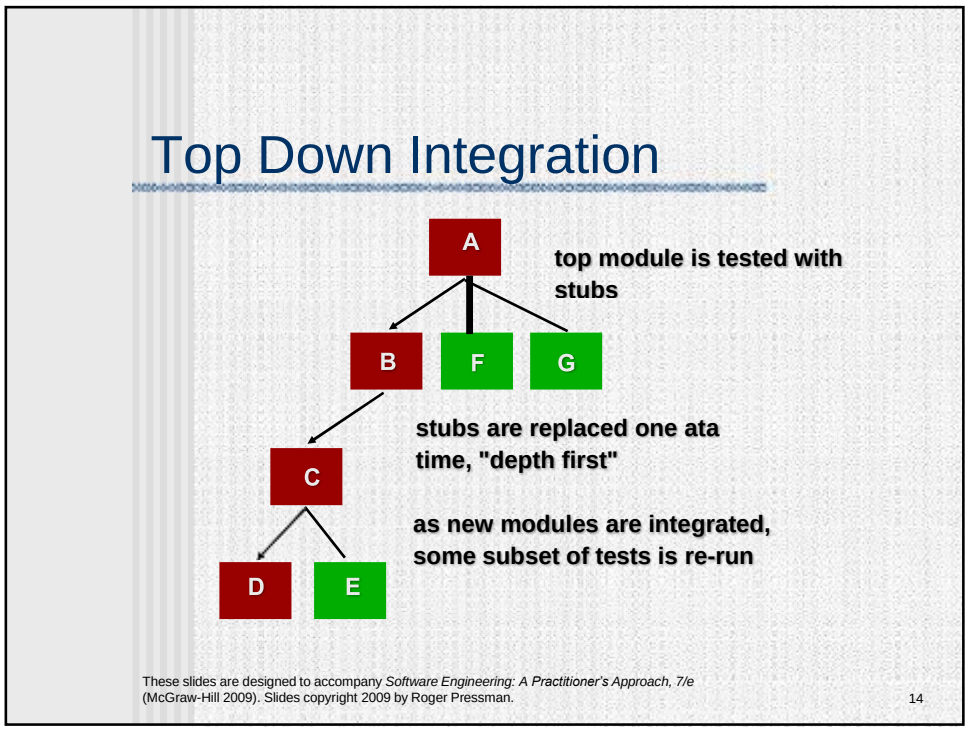
Incremental Testing Methodologies

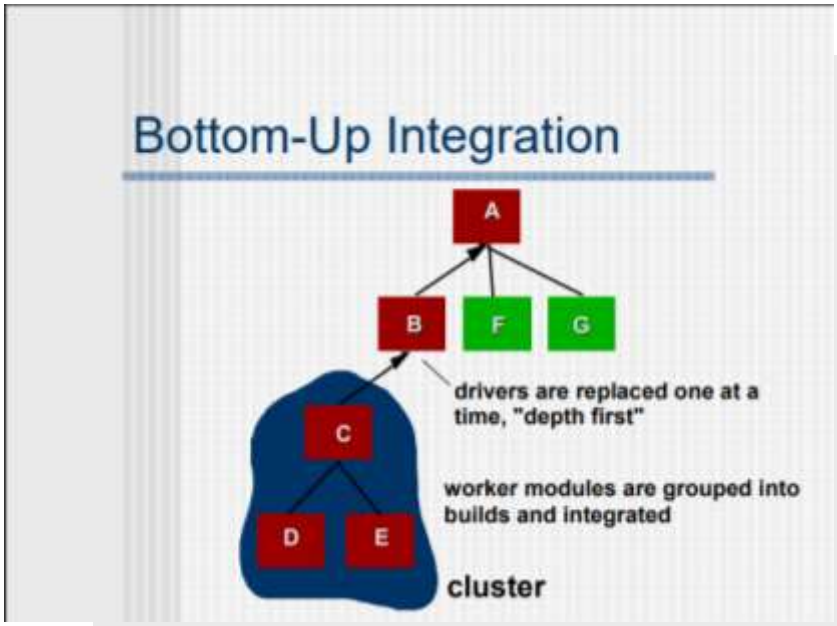
- **Top down Integration** - This type of integration testing takes place from top to bottom. Unavailable Components or systems are substituted by stubs
- **Bottom Up Integration** - This type of integration testing takes place from bottom to top. Unavailable Components or systems are substituted by Drivers
- **Functional incremental** - The Integration and testing takes place on the basis of the functions or functionalities as per the functional specification document.

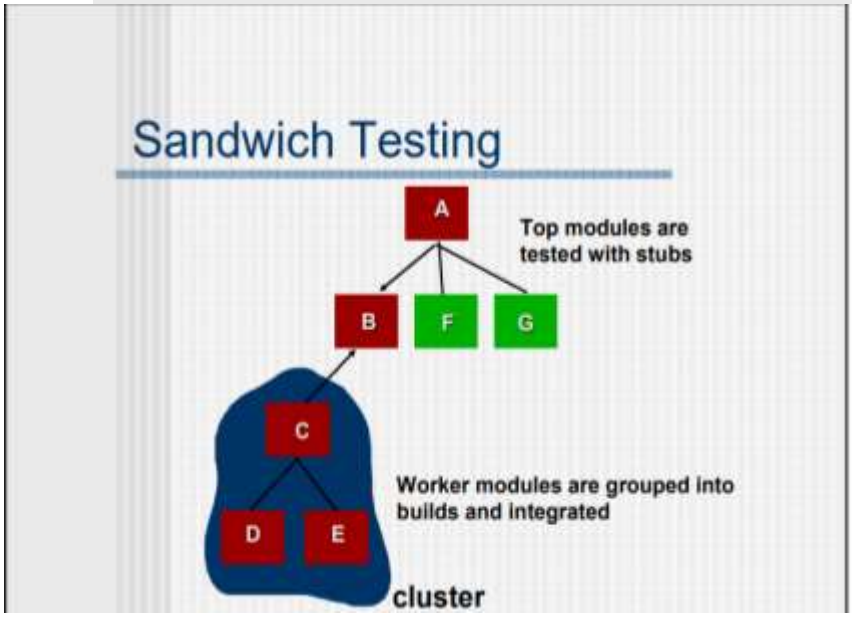
Incremental Testing - Features

- Each Module provides a definitive role to play in the project/product structure
- Each Module has clearly defined dependencies some of which can be known only at the runtime.
- The incremental integration testing's greater advantage is that the defects are found early in a smaller assembly when it is relatively easy to detect the root cause of the same.

- A disadvantage is that it can be time-consuming since stubs and drivers have to be developed for performing these tests.




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


Sandwich Testing
A
Top modules are tested with stubs
B
F
G
C
Worker modules are grouped into builds and integrated
D
E
cluster

Sandwich Testing

As discussed earlier, sandwich testing is a hybrid of the bottom-up and top-down approaches. Meaning, both top-level modules and lower-level modules are integrated and tested together as a system. Therefore, sandwich testing is also known as hybrid integration testing. This testing requires the use of both stubs and drivers.

How to run a sandwich testing?

The sandwich testing process can be divided into five steps:

- **Step 1** − Prepare the test plan
- **Step 2** − Develop the test scenarios, cases, and scripts
- **Step 3** − Run the test cases and report the defects
- **Step 4** − Track the defects and test them again
- **Step 5** − Repeat steps 3 and 4 until the integration is completed successfully

Attributes of Sandwich Testing

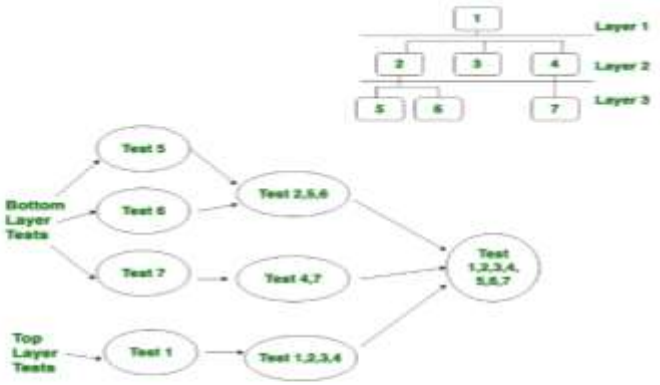
- Decide the methods or approaches to be used in the testing process
- Allocate scopes and out of scope items
- Assign roles and responsibilities
- Accommodate prerequisites for the testing
- Prepare the testing environment
- Prepare risk and mitigation plans

Before running a sandwich testing, it's crucial to study the architecture design of the application. This helps you identify the critical modules better. Testers can contact the architectural team to obtain the interface design to create and verify test cases and interface details.

There are 3 simple steps to perform sandwich testing which are given below.

1. Test the user interface in isolation using stubs.
2. Test the very lowest-level functions by using drivers.
3. When the complete system is integrated only main target (middle) layer remains for final test.

For **example:**



Advantages of Sandwich Testing:

- Sandwich Testing approach is used in very large projects having sub projects.
- It allows parallel testing.
- It is time saving approach.

Regression Testing

- *Regression testing* is the re-execution of some subset oftests 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 totesting 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 automatedcapture/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.

Sample Smoke Test Cases Example

T.ID	TEST SCENARIOS	DESCRIPTION	TEST STEP	EXPECTED RESULT	ACTUAL RESULT	STATUS
1	Valid login credentials	Test the login functionality of the web application to ensure that a registered user is allowed to login with username and password	1.Launch the application 2.Navigate the login page 3.Enter valid username 4.Enter valid password 5.Click on login button	Login should be success	as expected	Pass
2	Adding item functionality	Able to add item to the cart	1.Select categories list 2.Add the item to cart	Item should get added to the cart	Item is not getting added to the cart	Fail
3	Sign out functionality	Check sign out functionality	1. select sign out button	The user should be able to sign out.	User is not able to sign out	Fail

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).

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

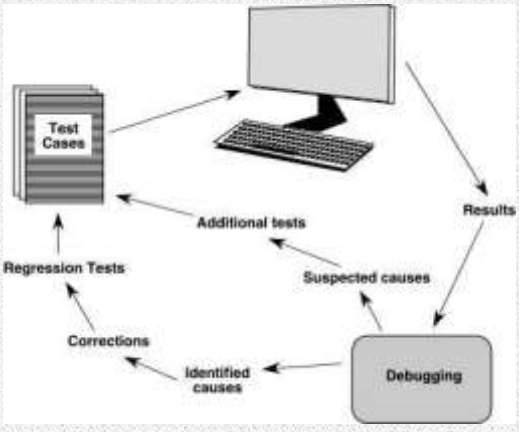
- Navigation throughout the architecture is tested.
- The WebApp is implemented in a variety of different environmental configurations and is tested for compatibility with each configuration.
- Security tests are conducted in an attempt to exploit vulnerabilities in the WebApp or within its environment.
- 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.

High Order Testing

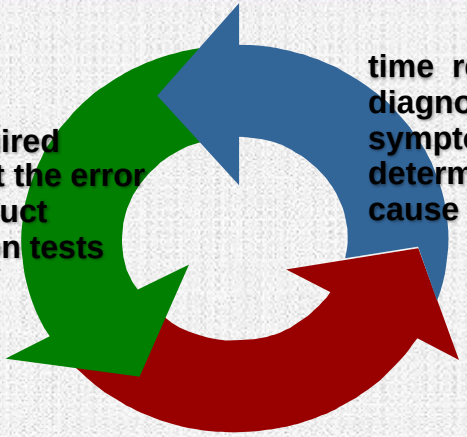
- Validation testing
  - Focus is on software requirements
- System testing
  - Focus is on system integration
- Alpha/Beta testing
  - Focus is on customer usage
- Recovery testing
  - forces the software to fail in a variety of ways and verifies that recovery is properly performed
- Security testing
  - 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

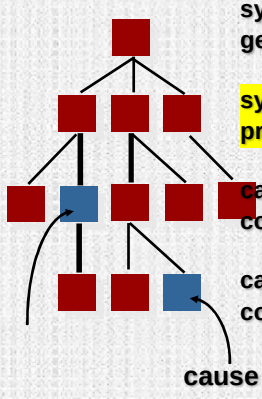
The Debugging Process



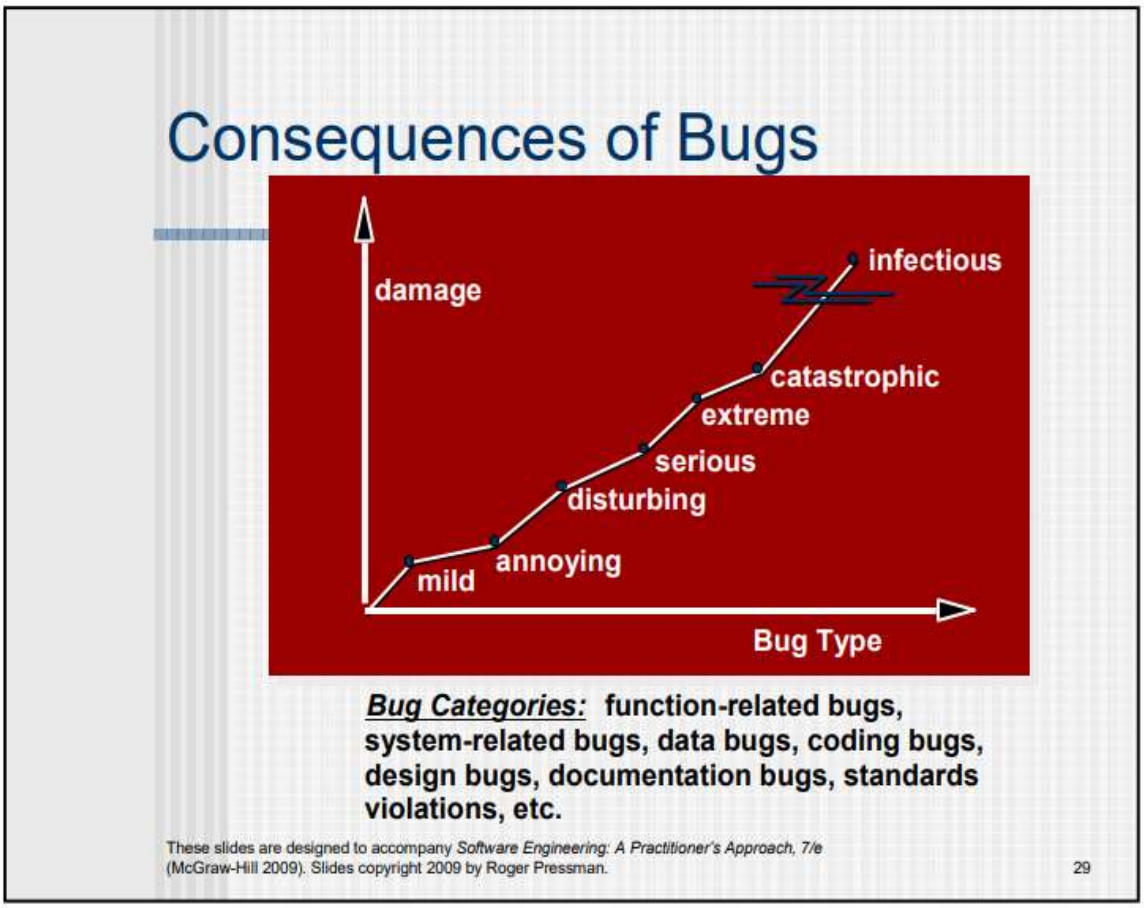
Debugging Effort

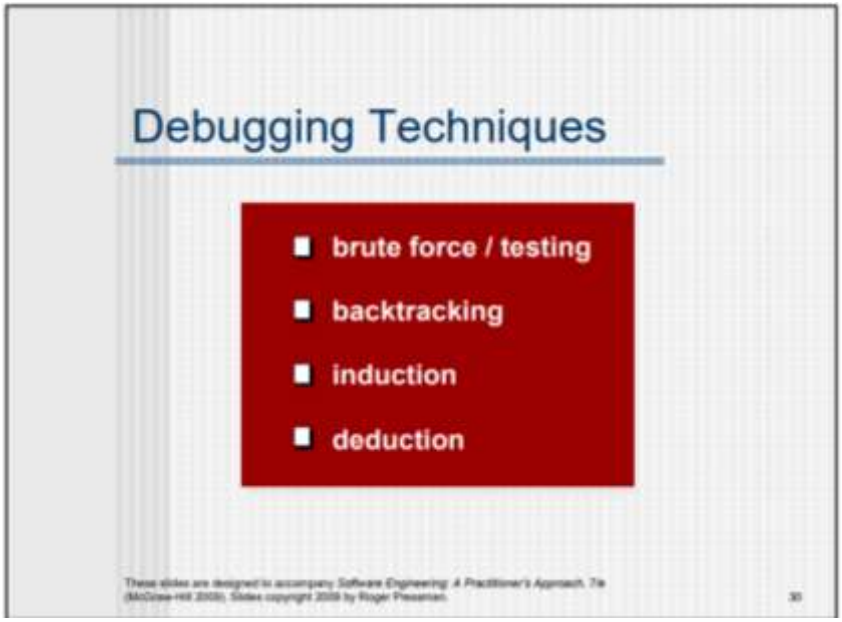


Symptoms & Causes



- symptom and cause may be geographically separated
- symptom may disappear whenanother problem is fixed
- cause may be due to a combination of non-errors
- cause may be due to a systemor compiler error
- cause may be due to assumptions that everyone believes
- symptom may be intermittent





Bruteforce: The brute force category of debugging is probably the most common and least efficient method for isolating the cause of a software error. You apply brute force debugging methods when all else fails. Using a "let the computer find the error" philosophy, memory dumps are taken, run-time traces are invoked, and the program is loaded with output statements(waste of time)

Backtracking: Beginning at the site where a symptom has been uncovered, the source code is traced backward (manually) until the cause is found

cause elimination—is manifested by induction or deduction and introduces the concept of binary partitioning. 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

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