

Chapter 22

■ Software Testing Strategies

Slide Set to accompany

Software Engineering: A Practitioner's Approach

by Roger S. Pressman

Slides copyright © 1996, 2001, 2005, 2009 by Roger S. Pressman

For non-profit educational use only

May be reproduced ONLY for student use at the university level when used in conjunction with *Software Engineering: A Practitioner's Approach*. Any other reproduction or use is prohibited without the express written permission of the author.

All copyright information MUST appear if these slides are posted on a website for student use.

Software Testing

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

“Microsoft is a testing company”
– Bill Gates

Table ES-1. Allocation of Effort

	Requirements Analysis	Preliminary Design	Detailed Design	<u>Coding and Unit Testing</u>	<u>Integration and Test</u>	<u>System Test</u>
1960s – 1970s	10%			80%	10%	
1980s	20%		60%		20%	
1990s	40%	30%		30%		

Source: Andersson, M., and J. Bergstrand. 1995. “Formalizing Use Cases with Message Sequence Charts.” Unpublished Master’s thesis. Lund Institute of Technology, Lund, Sweden.

The Economic Impacts of Inadequate Infrastructure for Software Testing, NIST Report 2002.

What Testing Shows

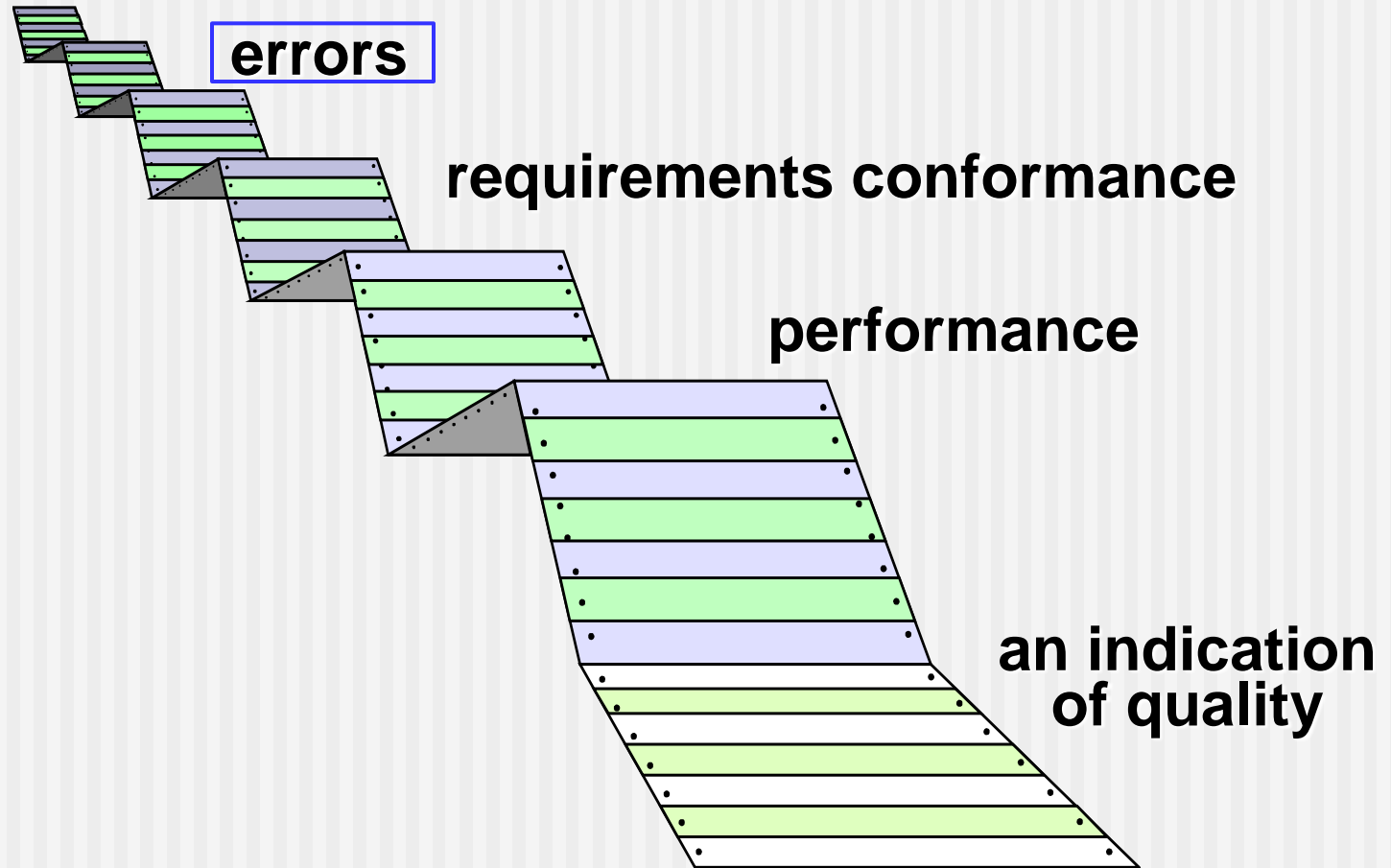


Table of Contents

- 22.1 A Strategic Approach to Software Testing
- 22.2 Strategic Issues
- 22.3 Test Strategies for Conventional Software
- 22.4 Test Strategies for Object-Oriented Software
- 22.5 Test Strategies for WebApps
- 22.6 Test Strategies for MobileApps
- 22.7 Validation Testing
- 22.8 System Testing
- 22.9 The Art of Debugging

22.1 A Strategic Approach to Software Testing

- To perform effective testing, you should conduct **effective technical reviews prior to testing**.
 - Many errors will be eliminated before testing commences.
- Testing begins at the component level (**unit testing**) and works "outward" toward the integration of the entire computer-based system (**integration testing** and **system testing**).
- 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**.
- **Debugging** must follow after testing.

22.1.1 Verification & Validation

- **Verification:** the set of tasks that ensure that software correctly implements a specific function.
- **Validation:** the set of tasks that ensure that the software that has been built is traceable to customer requirements.
- Boehm [Boehm 81] states this another way:
 - **Verification:** "Are we building the product right?"
 - **Validation:** "Are we building the right product?"
 - ☛ Implies that “validation” is “requirements validation”.
 - ☛ A better way to distinguish them would be
 - Verification requires a verification perspective.
 - Validation has an established set of perspectives.

22.1.2 Organization for Software Testing

Is it better to
have an
independent
tester?



developer

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

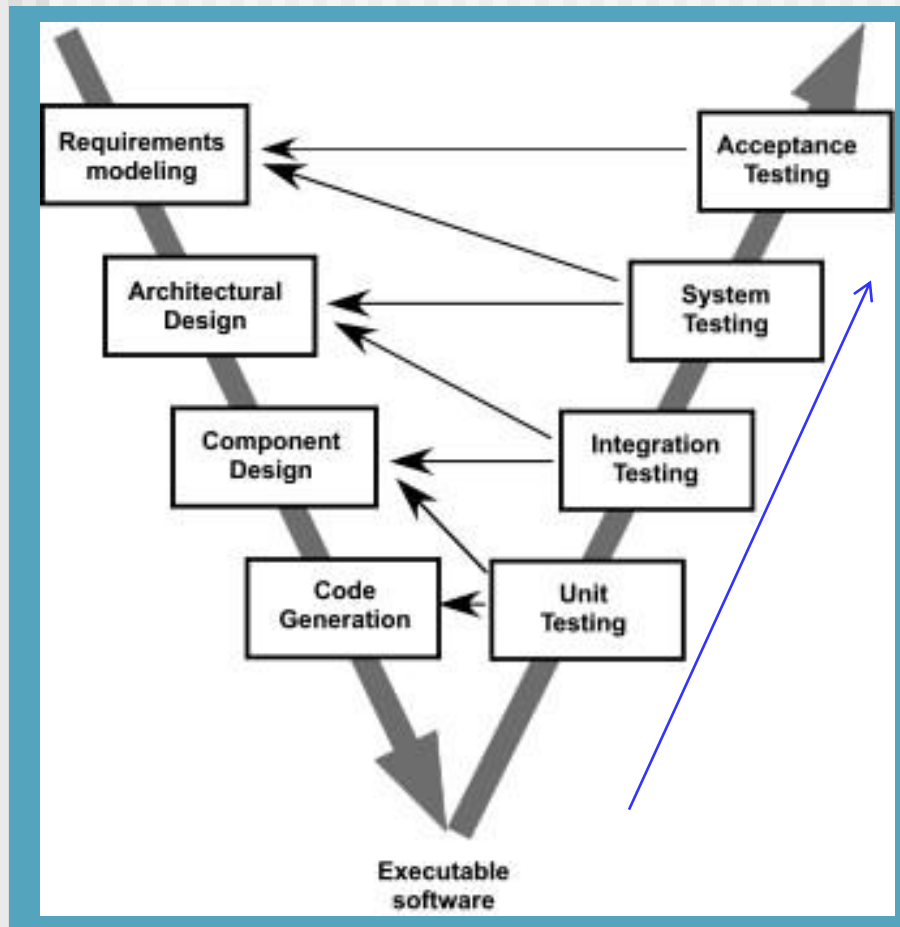


independent tester

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

22.1.3 Software Testing Strategy – The Big Picture

The V-Model



Quality assurance model has been added to the classic life cycle

Testing Strategy

- We begin by 'testing-in-the-small' and move toward 'testing-in-the-large'
- For conventional software
 - Modules (components) --> Integration of modules
- For OO software
 - OO classes --> Integration of classes
 - For unit testing, however, a single operation can no longer be tested in isolation (the conventional view), but rather as part of a class.

Example

Consider an operation X that is defined for the superclass. X is used in different subclasses in subtly different ways.

☛ Should test X in the context of each subclass.

22.1.4 Criteria for Completion of Testing

“Program testing can at best show the presence of **errors** but never their **absence**.” – E. W. Dijkstra

- By **collecting metrics** during software testing and making use of existing software **reliability models**, it is possible to determine “When are we done testing?”
- **Further work remains to be done** to establish quantitative rules for testing.
- But the current empirical approaches are considerably better than raw intuition.

22.2 Strategic Issues

Tester should know almost as much as the developer

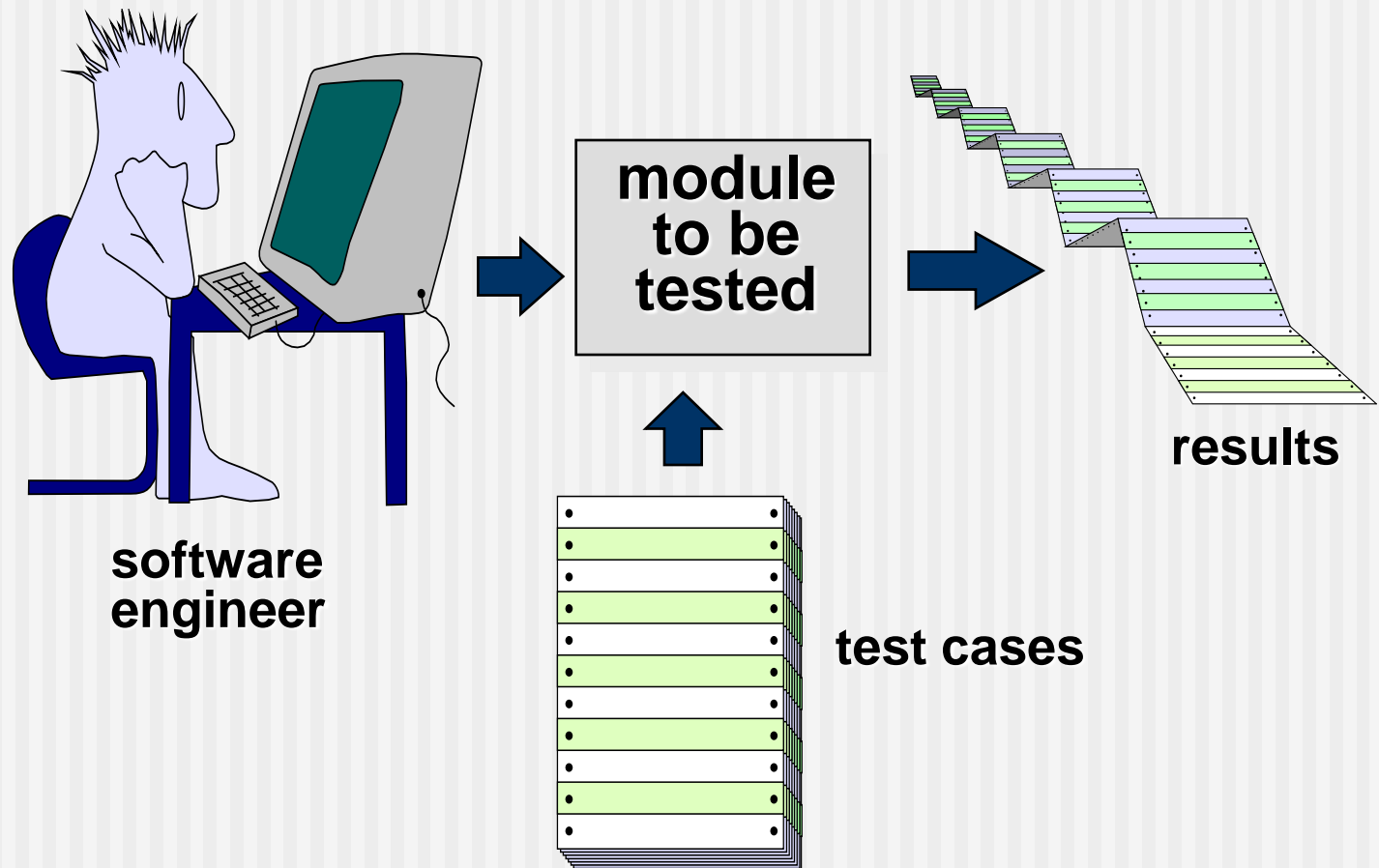
- Before testing starts:

- Specify product requirements in a quantifiable manner.
- 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

- After testing starts:

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

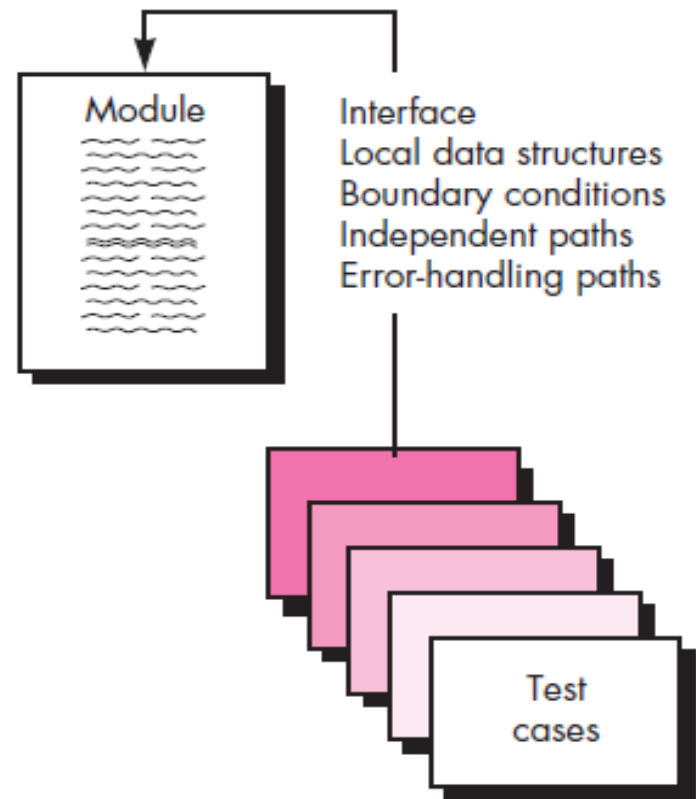
22.3 Test Strategies for Conventional Software



22.3.1 Unit Testing

FIGURE 17.3

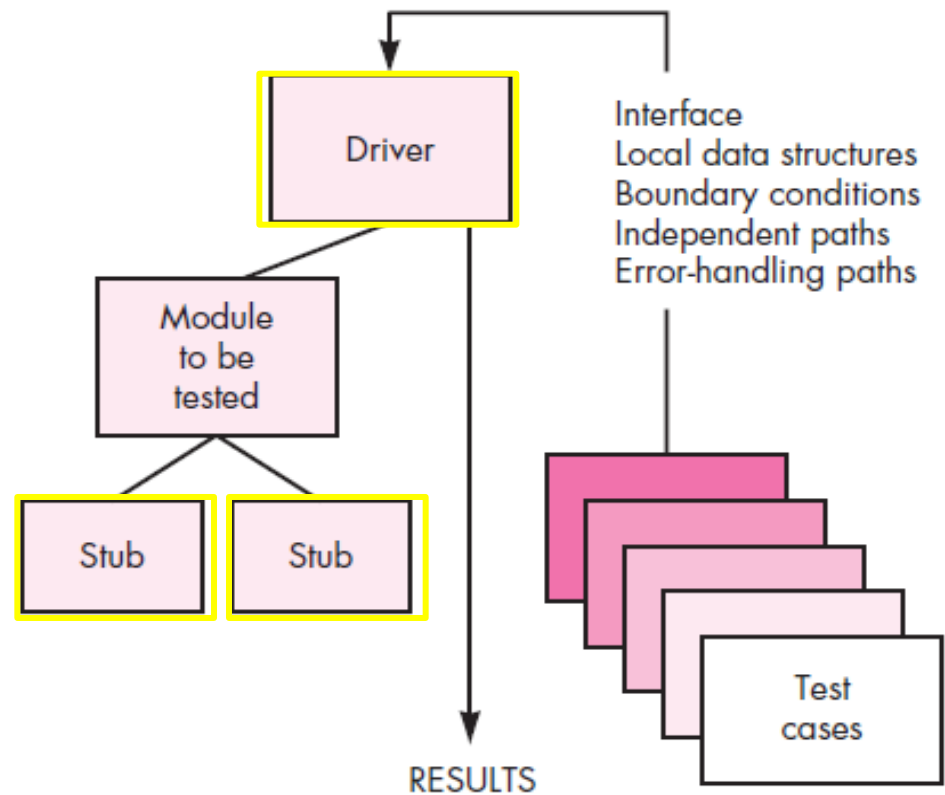
Unit test



Unit Test Environment

FIGURE 17.4

Unit-test
environment

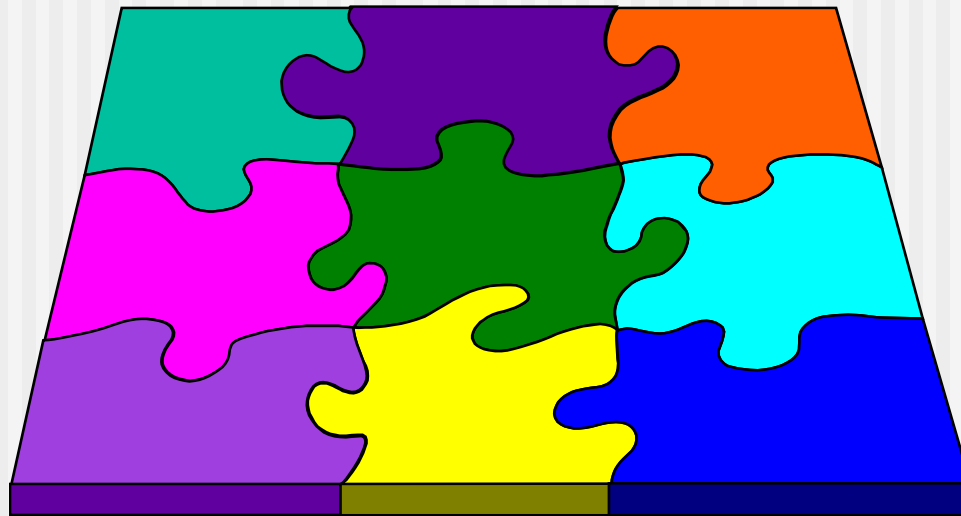


22.3.2 Integration Testing

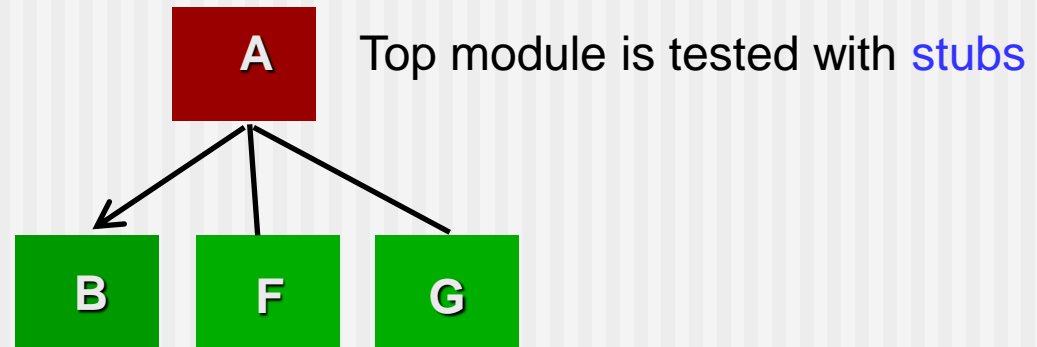
Options:

- The “big bang” approach
- An incremental strategy

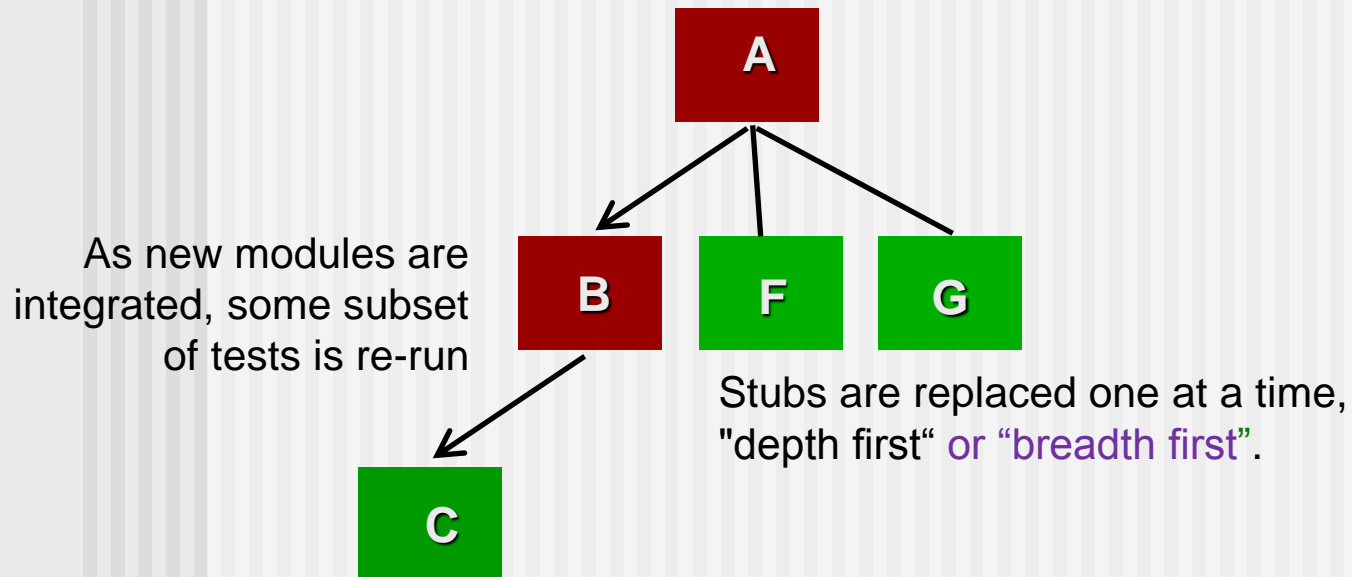
What is the problem with the big bang approach?



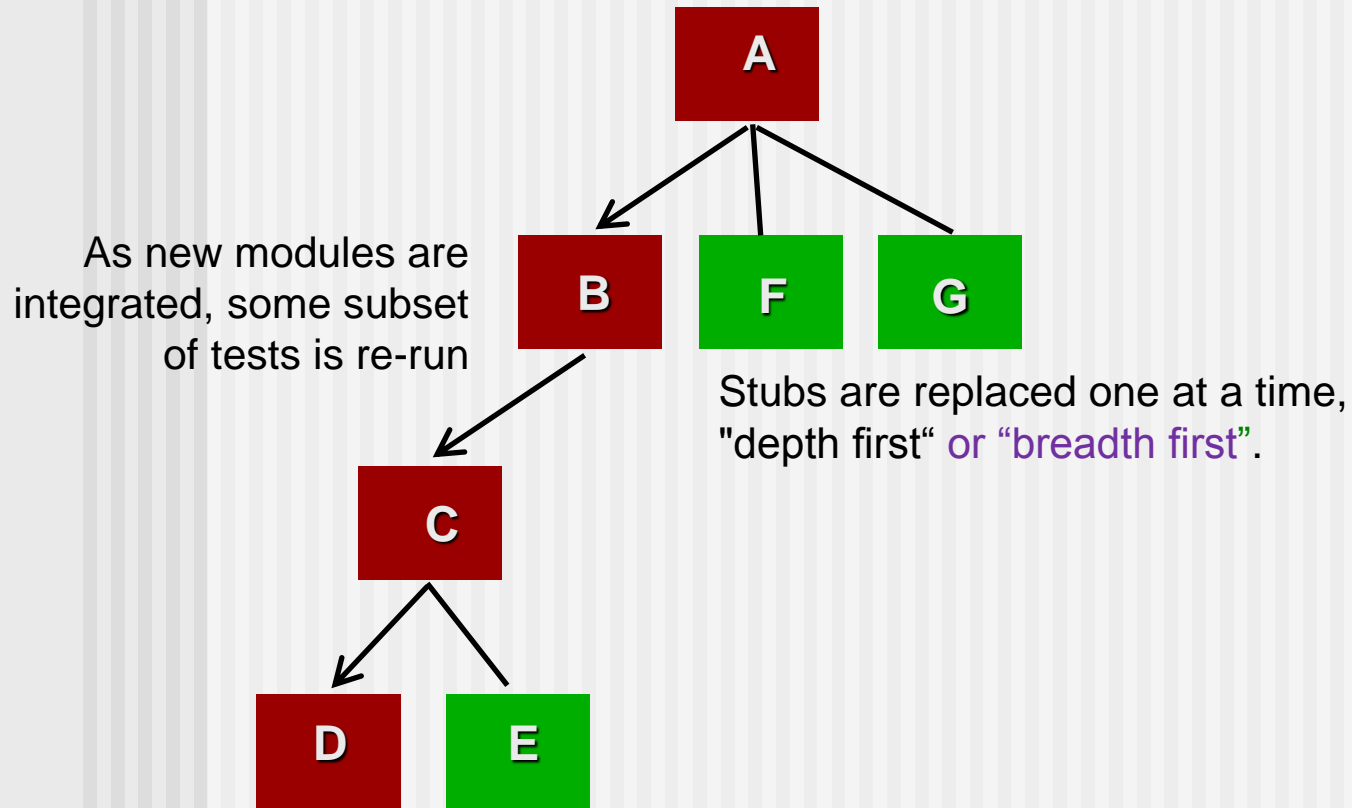
Top Down Integration(1/3)



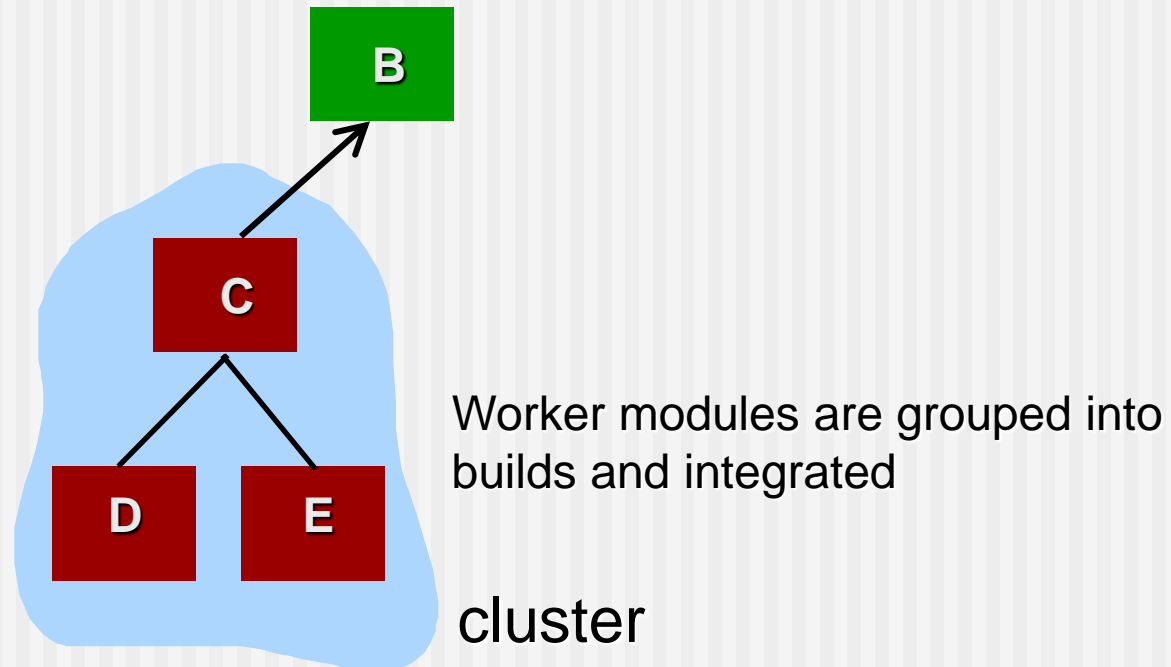
Top Down Integration(2/3)



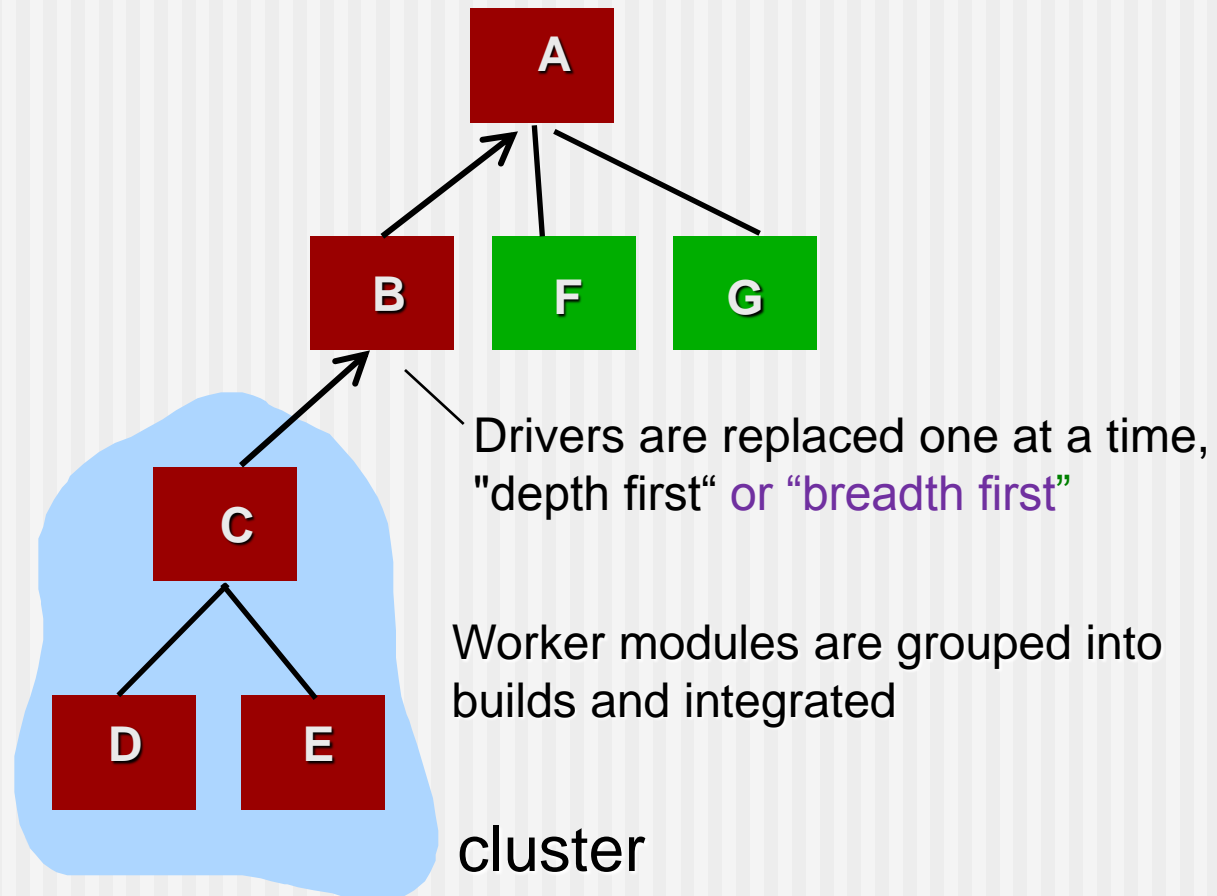
Top Down Integration(3/3)



Bottom-Up Integration (1/2)

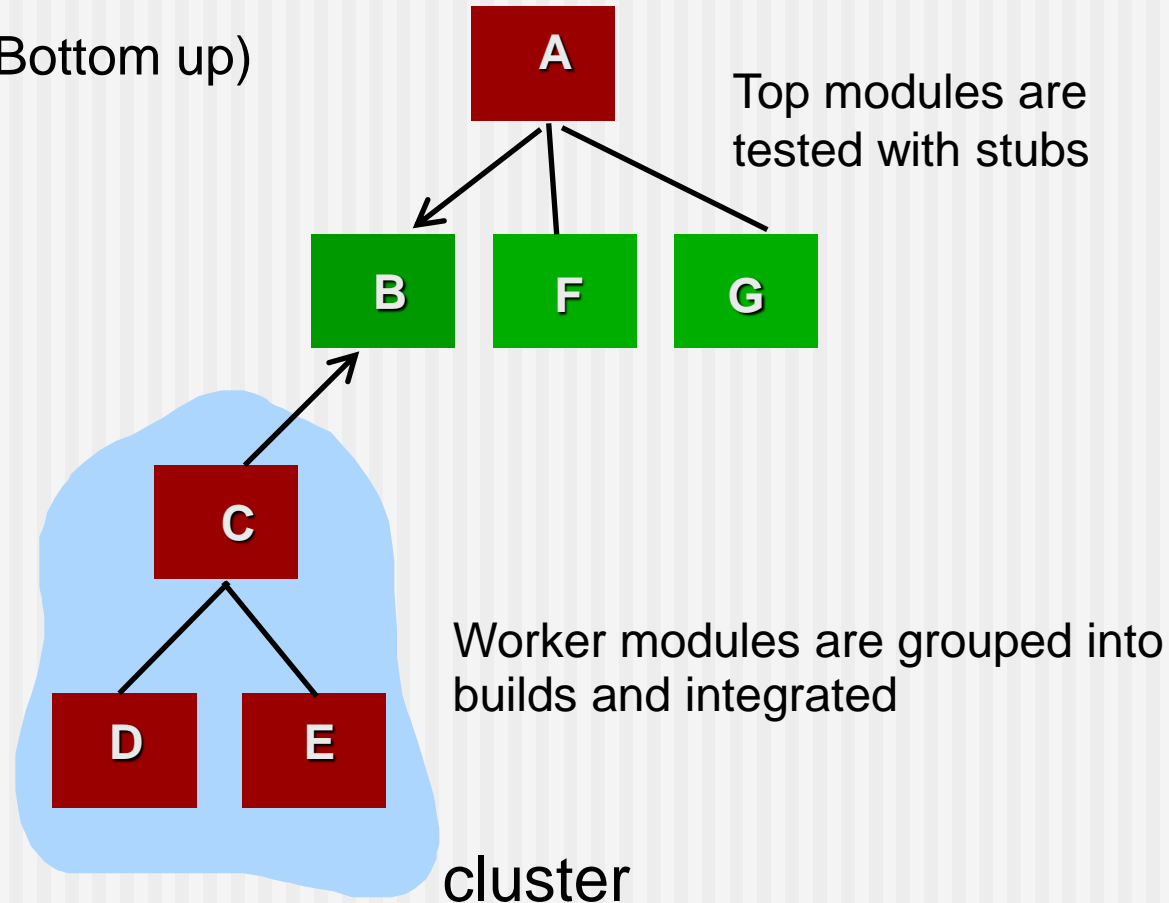


Bottom-Up Integration (2/2)

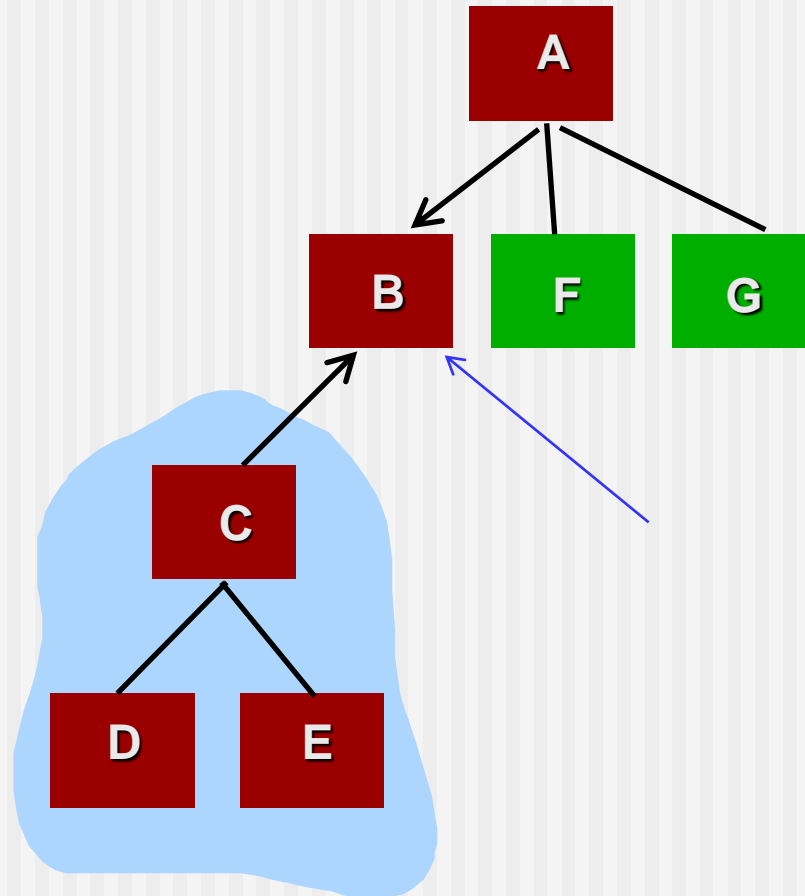


Sandwich Testing (1/2)

(= Top Down Bottom up)



Sandwich Testing (2/2)



Regression Testing

- Re-execution of a subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects
- 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
 - To uncover the errors that have the highest likelihood of throwing the software project behind schedule.
- Steps:
 1. Integrate software components 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.
 2. Design tests to expose errors in the build
 - A subset of regression test cases
 3. Integrate the build with other builds

22.4 Test Strategies for OOTesting

- Before testing starts
 - Review the analysis and design models for their correctness and consistency
- After testing starts
 - Testing strategy changes from the conventional testing
 - 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

Review the OOAD models

- It is useful because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code level.
- ☛ An uncovered problem in the definition of class attributes will circumvent side effects that might occur if the problem were not discovered until design or code.

Review the CRC Model (Before testing starts)

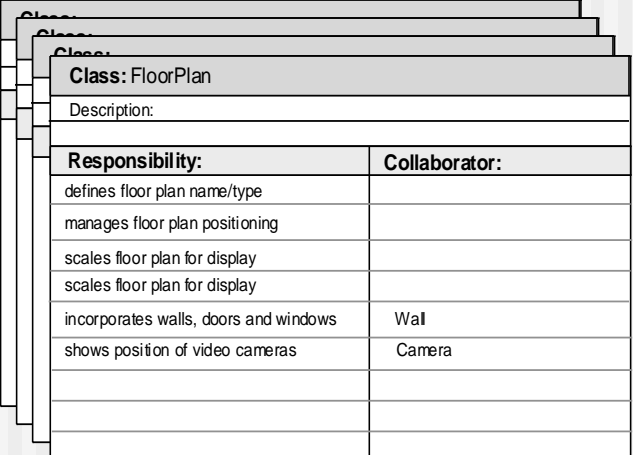
1. Revisit the CRC model and the object-relationship model.
2. Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator's definition.
3. Invert the connection to ensure that each collaborator that is asked for service is receiving requests 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.

22.4.1 Unit Testing in the OO Context

- Class testing is the equivalent of unit testing
 - operations within the class are tested
 - the state behavior of the class is examined

22.4.2 Integration Testing in the OO Context

- 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**



Class: FloorPlan	
Description:	
Responsibility:	Collaborator:
defines floor plan name/type	
manages floor plan positioning	
scales floor plan for display	
scales floor plan for display	
incorporates walls, doors and windows	Wall
shows position of video cameras	Camera

22.7 Validation testing

- Focus is on **conformance to** the software requirements **as a whole**
 - ➔ Validation testing is a **system testing in a wide sense**
- Validation-Test Criteria
 - A test plan outlines the kinds of tests to be conducted
 - A test procedure defines specific test cases
- Configuration Review
 - Ensure that all elements of the software configuration have been properly developed, are cataloged, and have the necessary detail
- Alpha/Beta testing
 - **Alpha test**
 - Conducted at the developer's site
 - **Beta test**
 - Conducted at one or more end-user sites.
 - A variation - **customer acceptance testing**

22.8 System testing

- Focus is on system integration
- Kinds:
 - **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

22.9 The Art of Debugging

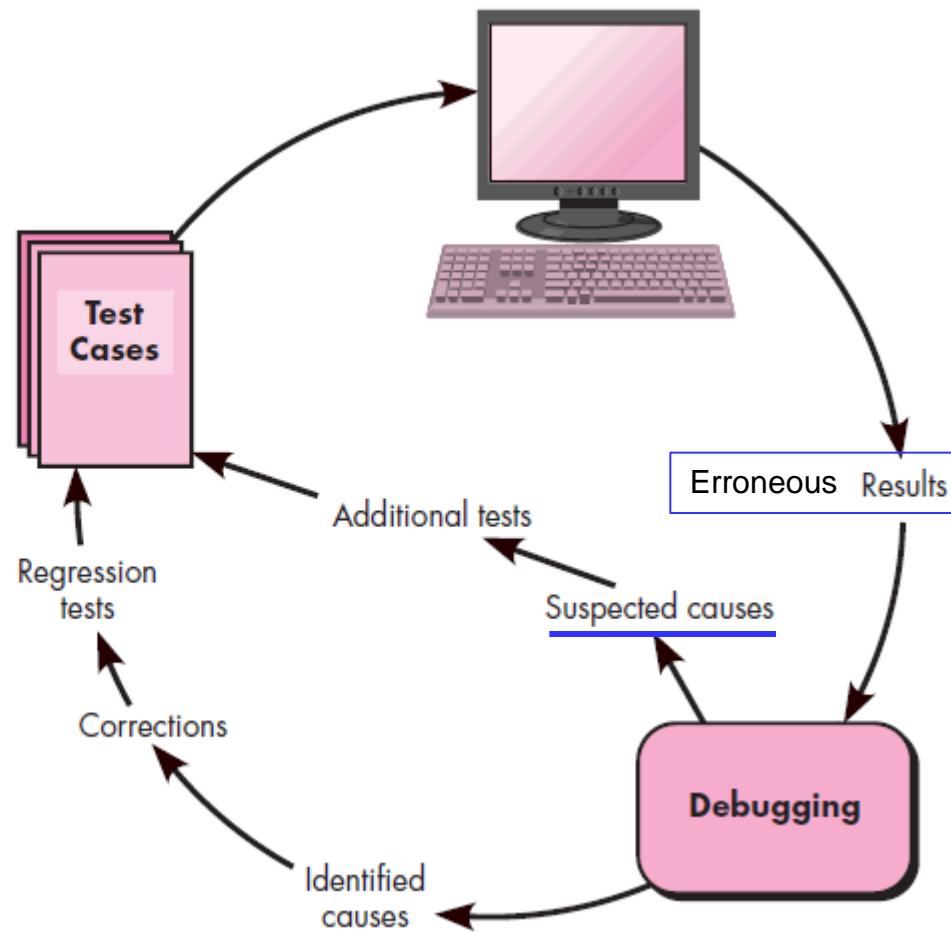


After a test case uncovers an error, debugging is the activity of removing the cause of the error.

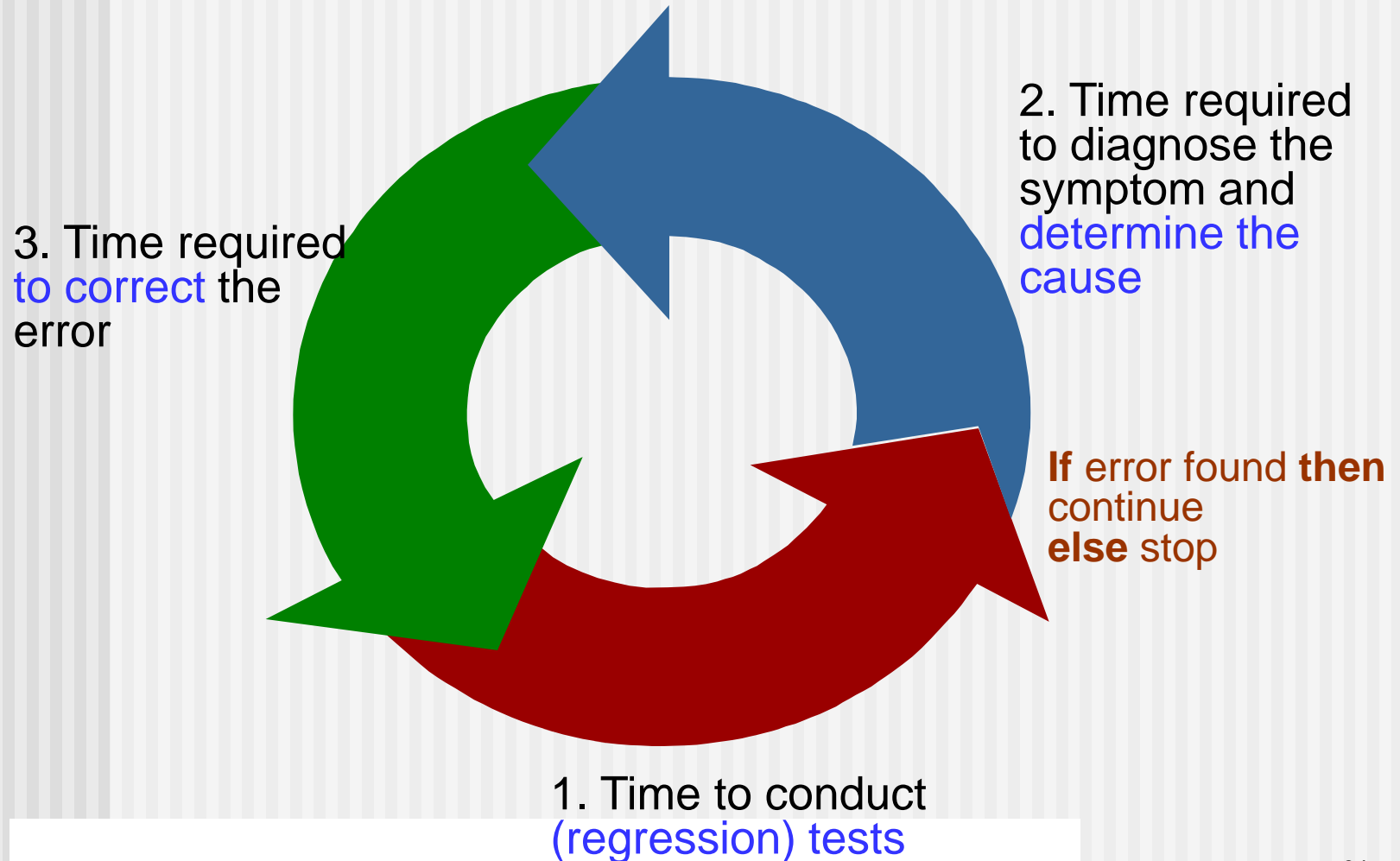
22.9.1 The Debugging Process

FIGURE 17.7

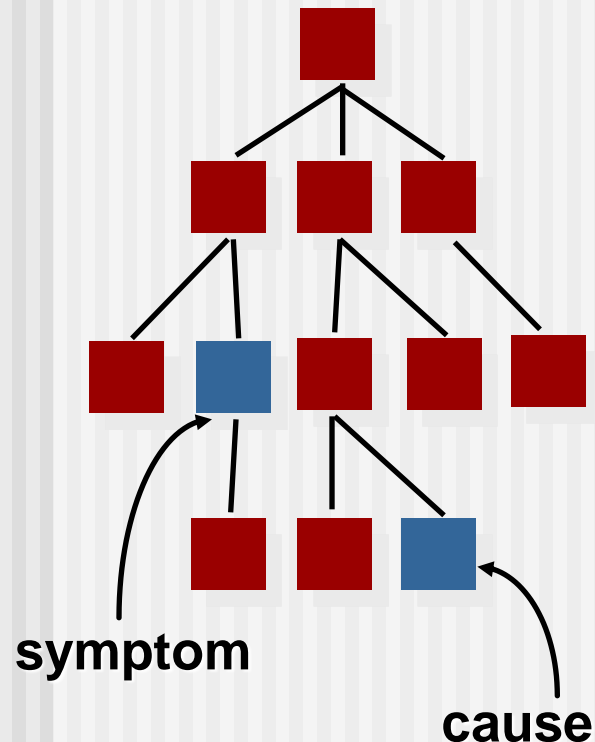
The debugging process



Debugging Effort

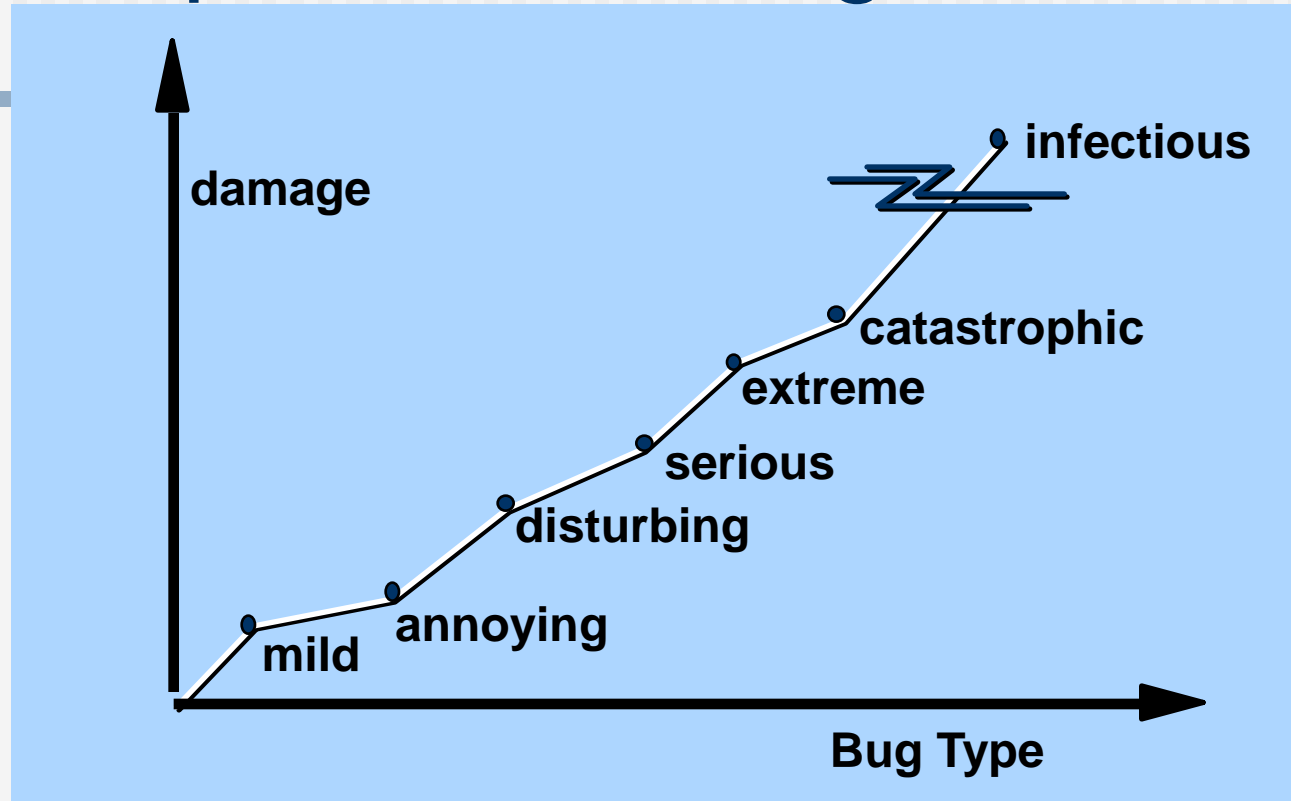


Symptoms & Causes



- ❑ Symptom and cause may be geographically separated
- ❑ Symptom may disappear when another problem is fixed
- ❑ Cause may be due to a 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.

22.9.2 Psychological Considerations

- Some people are good at debugging and others aren't.
- It is difficult to learn "debugging" but see the next section.

22.9.3 Debugging Strategies

Debugging Techniques

1. Brute force

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.

2. Backtracking

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

3. Cause Elimination

Induction

“Cause hypothesis” is devised and data is created to prove or disprove it. Repeat this until the cause is found.

Deduction

A list of all possible causes is developed, test is conducted to eliminate each.

■ Automated Debugging

- "... many new approaches have been proposed and many commercial debugging environments are available .

Integrated development environments (IDEs) provide a way to capture some of the language-specific predetermined errors (e.g., missing end-of-statement characters, undefined variables, and so so) without requiring compilation."

- However, tools are not a substitute for careful evaluation based on a complete design model and clear source code.

■ The People Factor

- When all else fails, get help (from other people)!

22.9.4 Correcting the Error

1. 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.
2. 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.
3. What could we have done to prevent this bug in the first place?
 - The first step toward establishing a statistical **SQA** 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