605.601 Foundations of Software Engineering
Fall 2020

Module 07: Software Testing

Dr. Tushar K. Hazra

tkhazra@gmail.com

(443)540-2230

605.601 Foundations of Software Engineering
Course Module 07: Testing

Topics for Discussion

- Introduction - Concepts
- Software Testing Strategies
- Different Types of Testing
- Black and White Box Testing

What is Testing?

- Testing is **focused on finding faults**
  - Discovery process to prevent failures
  - Focus is fault correction rather than assigning blame

Fault

- "A **physical defect or flaw** within a hardware or software component" [Rogers, 2009]

Error

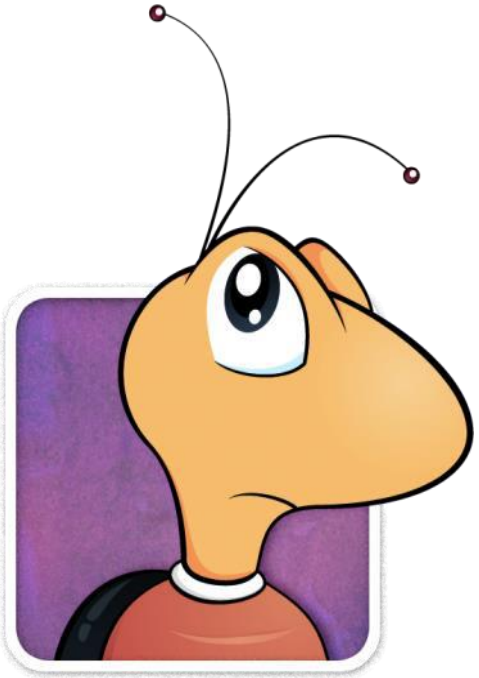
- "The **manifestation of a fault**: a deviation from accuracy or correctness in state" [Rogers, 2009]

Failure

- An **externally-visible** (e.g., user-visible) event representing a deviation from the system's required behavior [Pfleeger and Atlee, 2006; Rogers, 2009]

What about bugs?

- The term "bug" suggests that **faults originate from external sources**
- Faults originate from. . .
  - **Missing** or unachievable **requirements**
  - **Incorrect specification, design, or implementation**



Bug OR No Bug …

- [We should clean] up our language by no longer calling a bug a bug but by **calling it an error**.
- It is much more honest because it squarely puts the blame where it belongs, viz. with the **programmer who made the error**.
- The animistic metaphor of the bug that **maliciously sneaked in while the programmer was not looking** is intellectually dishonest as it disguises that the error is the programmer's own creation. [Dijkstra, 1988]

Verification & Validation

- 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 (1981) states it like this:
  - .., **Verification**: "**Are we building the product right**?"
  - .., **Validation: "Are we building the right product**?"

Course Module 07: Testing

- Testing Strategy: What it may entail?
  - **Use code reviews**
    - .., Effective at finding algorithmic, design, documentation, etc. faults
  - Start with **individual components**
  - Use **independent test team**
    - .., Removes conflict between responsibility for fault and need to discover as many faults as possible
    - .., Allows additional interpretations of design, implementation, and documentation
    - .., Proceeds in parallel with development

Course Module 07: Testing

- Who Tests the Software?

Figure: A developer understands the system but will test "**gently**" and is driven by "delivery."

Image source: https://xkcd.com/676/

Figure: An independent tester must learn about the system but **will attempt to break it** and is driven by quality.

Image source: https://xkcd.com/979/

Test Plan

- Document that **describes how to show customer(s) that the software works correctly** (i.e., it is free of faults and satisfies requirements)
  - Establishes test objectives and organization of tests
  - Identifies criteria used to determine when testing is complete
  - Describes method(s) used to perform tests
  - Enumerates test cases for each test method
    - .., Includes how test cases will be generated
- The test plan provides a complete picture of how and why testing will be performed

Test Organization

Unit Test each component in isolation from other components

Integration Verification that multiple components work together as described in specification

System Function Verification that the system performs the functions described by requirements

Performance Verification that the system's hardware and software performs correctly (e.g., stress, recovery, and security testing)

Acceptance System checked against customers' requirements

Installation System performs correctly following installation in environment

Course Module 07: Testing

Testing in Progress …

- Unit Testing
- Integration Testing
- Regression Testing
- Debugging

Keep in Mind …

- Test Case - A particular choice of input data used when testing a program
- Test - A finite collection of test cases

Test Cases

- **Objective** is to separate input data into equivalence classes
- Classes should cover the **entire set of input data**
- Classes should be **disjoint** (i.e., particular input belongs to exactly one class)
- **Any element of class represents all elements of class**
  - .., If the code contains a fault, then that fault can be detected using any element of the class

- **Unit Test Environment**

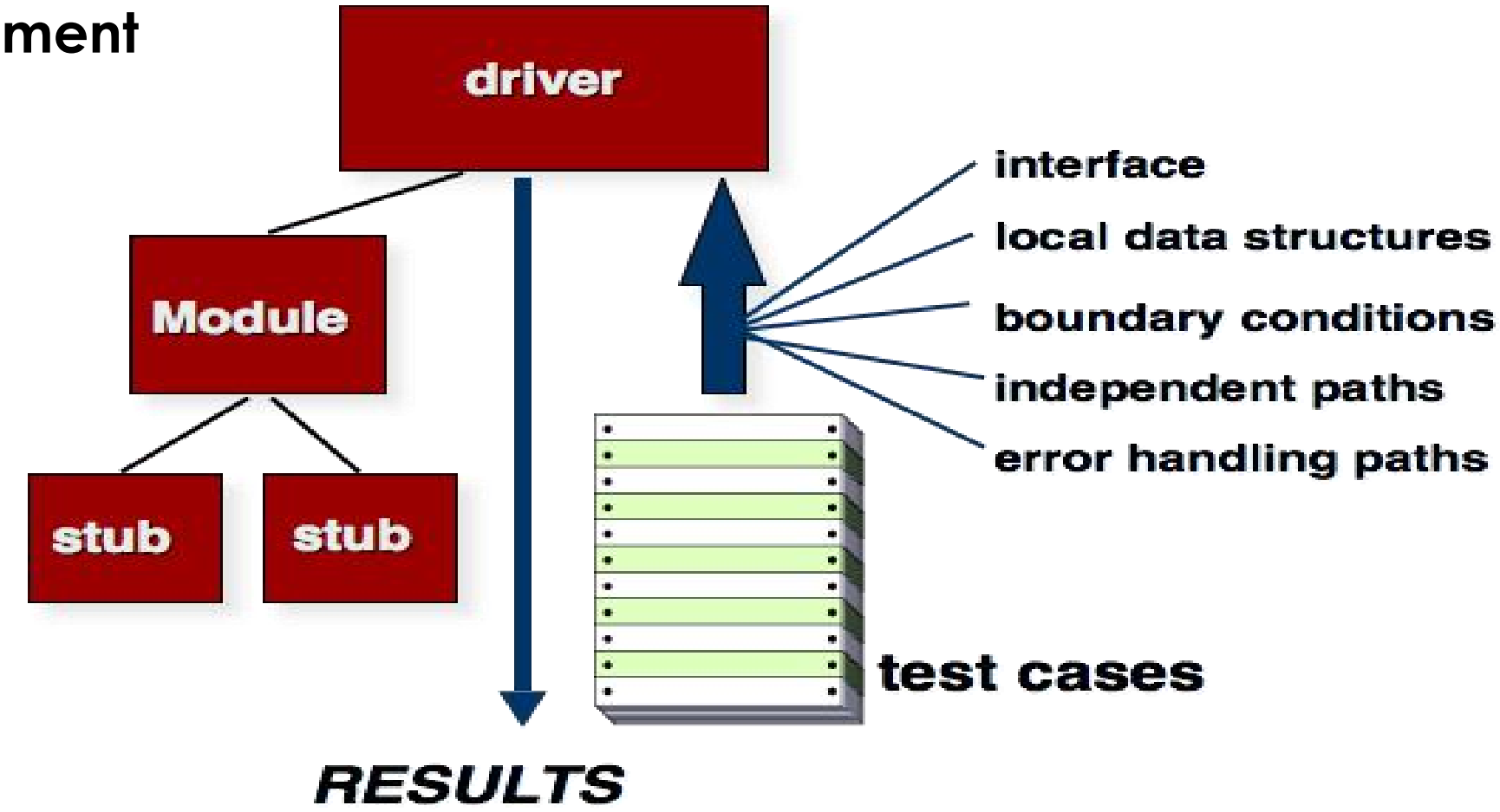


Image source: Pressman, R. (2010). *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., New York, NY, 7th edition

Driver

- A special program that **calls module with test case**

- Sometimes known as **a test runner** that automates the execution of tests and generating test reports
  - Most unit testing frameworks (e.g., x Unit) have pre-existing test runners

Course Module 07: Testing

- **Stubs**
  - A special component that **simulates a missing module**
  - Returns expected output for fixed input
  - Example
    - Remove an unnecessary test dependency (e.g., a database)

```
def has_failing_grade():
    # get scores from DB scores =
    get_scores()  for score in scores:
        if score < 70:
            return True
    return False
```

```
def test_has_failing_grade():
    global get_scores
    get_scores = lambda: \
        [70, 82, 67, 94]
    assert has_failing_grade()
    # TODO: Add test cases
```

Integration Testing

- The test plan explains why and how components are combined to test the complete system
  - Impacts timing of integration and order of implementing components
- **Strategies**
  - **Big bang**
  - **Bottom-up**
  - **Top-down**
  - **Sandwich**

- **Big Bang Integration**
  - Test components in isolation (maybe), combine to create complete system, and see if it works!
  - **Not recommended for any system**
    - Requires both drivers and stubs to test components
    - Difficult to find cause of failures
    - Not easy to distinguish interface faults from other faults

Image source: https://en.wikipedia.org/wiki/File:The_Big_Bang_Theory_(Official_Title_Card).png

Bottom Up Integration

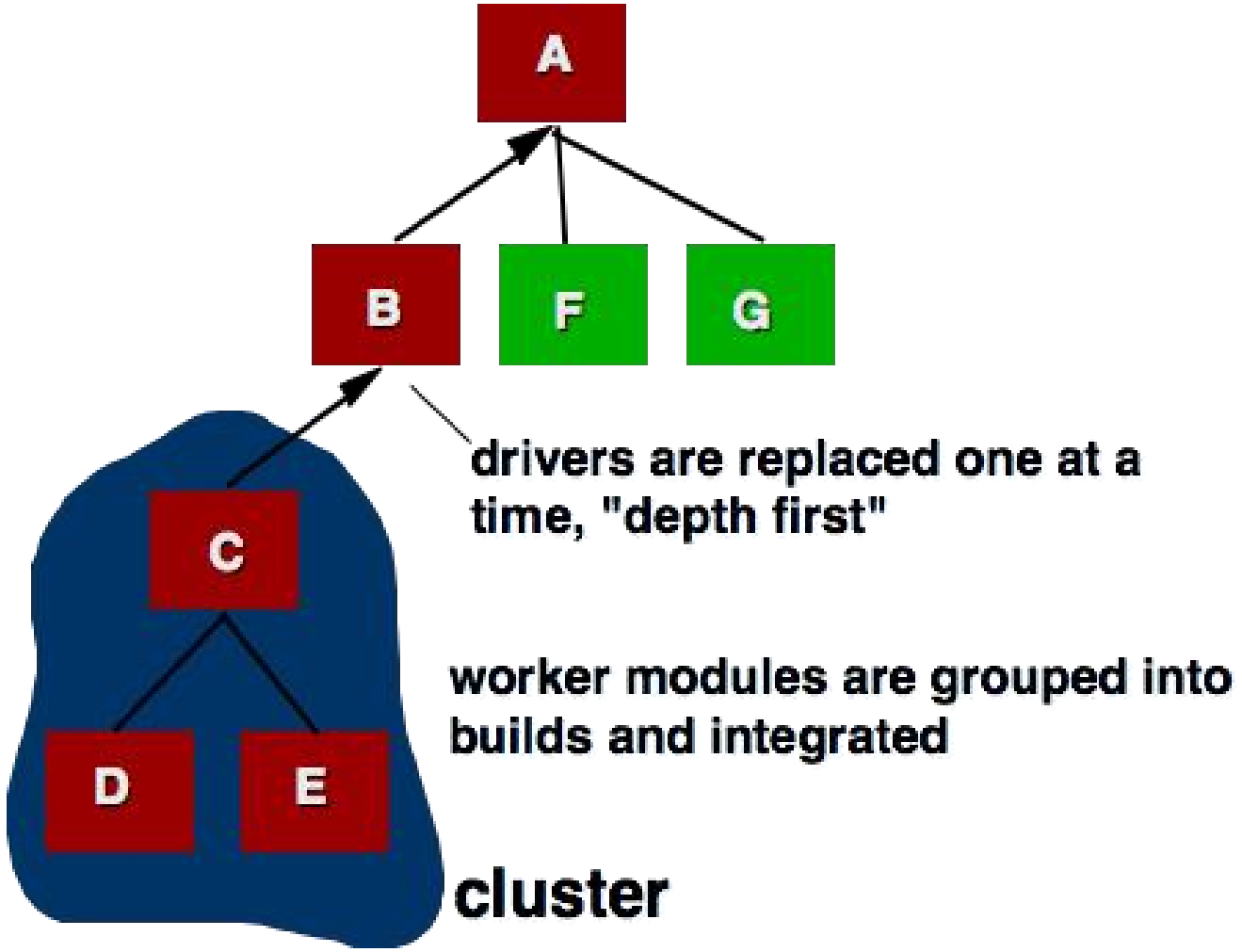


Image source: Pressman, R. (2010). *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., New York, NY, 7th edition

Bottom-Up Integration

- Process
1. **Test** each (low-level) component individually using drivers
2. **Replace** drivers with higher-level components
3. **Continue** until all components integrated
- Notes
- Good match for many **object-oriented programs**
- Top-level components typically the most complex but tested last
  - .., May delay finding major faults until the end of testing

Image source: https://commons.wikimedia.org/wiki/File:
Louis-Emile_Durandelle,_The_Eiffel_Tower_-_State_of_the_Construction,_1888,_grayscale.jpg

- **Top Down Integration**

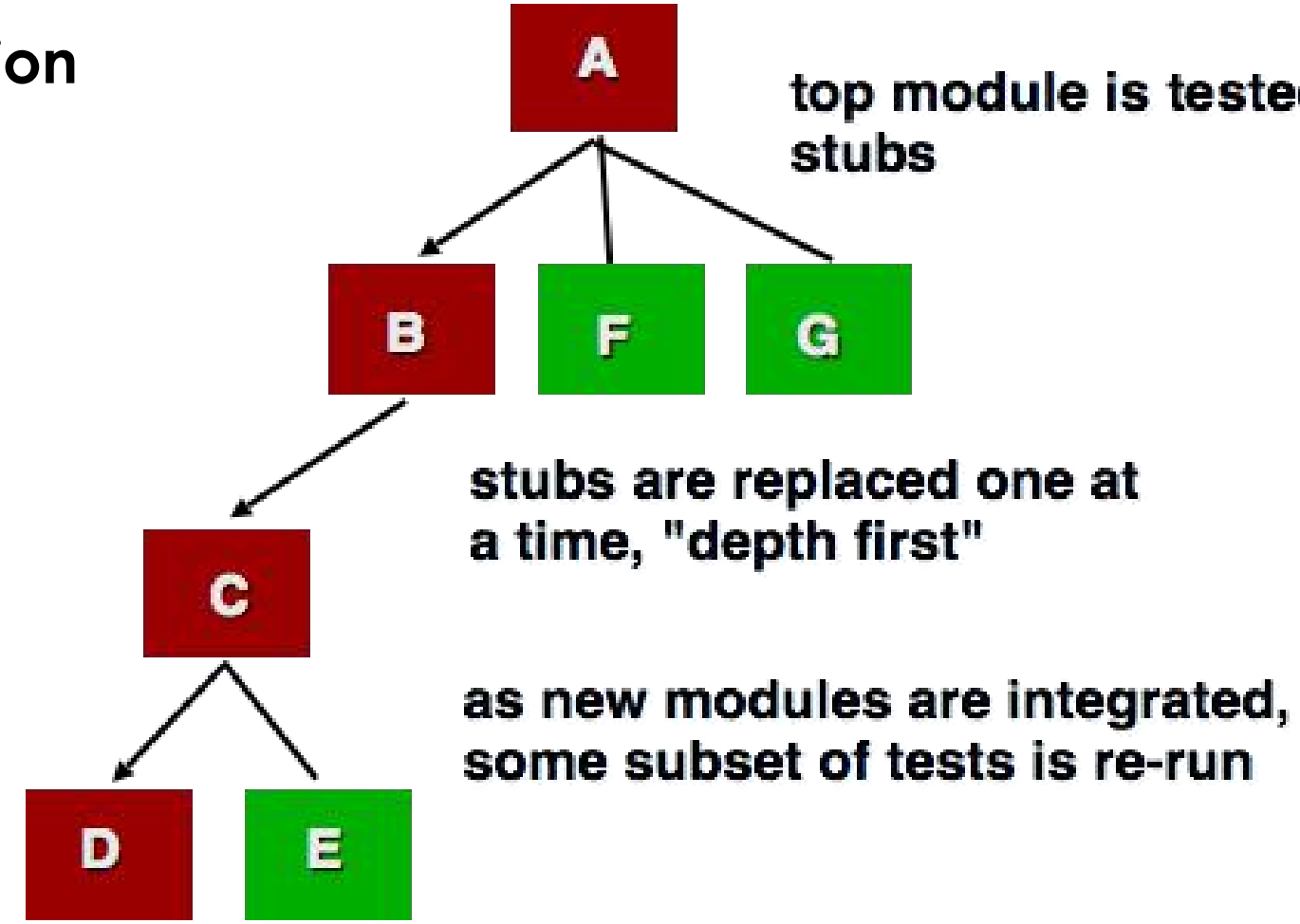


Image source: Pressman, R. (2010). *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., New York, NY, 7th edition

Top-Down Integration

- Process

1. **Test** top-level component(s) individually using stubs
2. **Replace** each stub with actual component and retest
3. **Continue** until all components integrated

- Disadvantages
  - May require a large number of stubs for lower-level components

Sandwich Testing

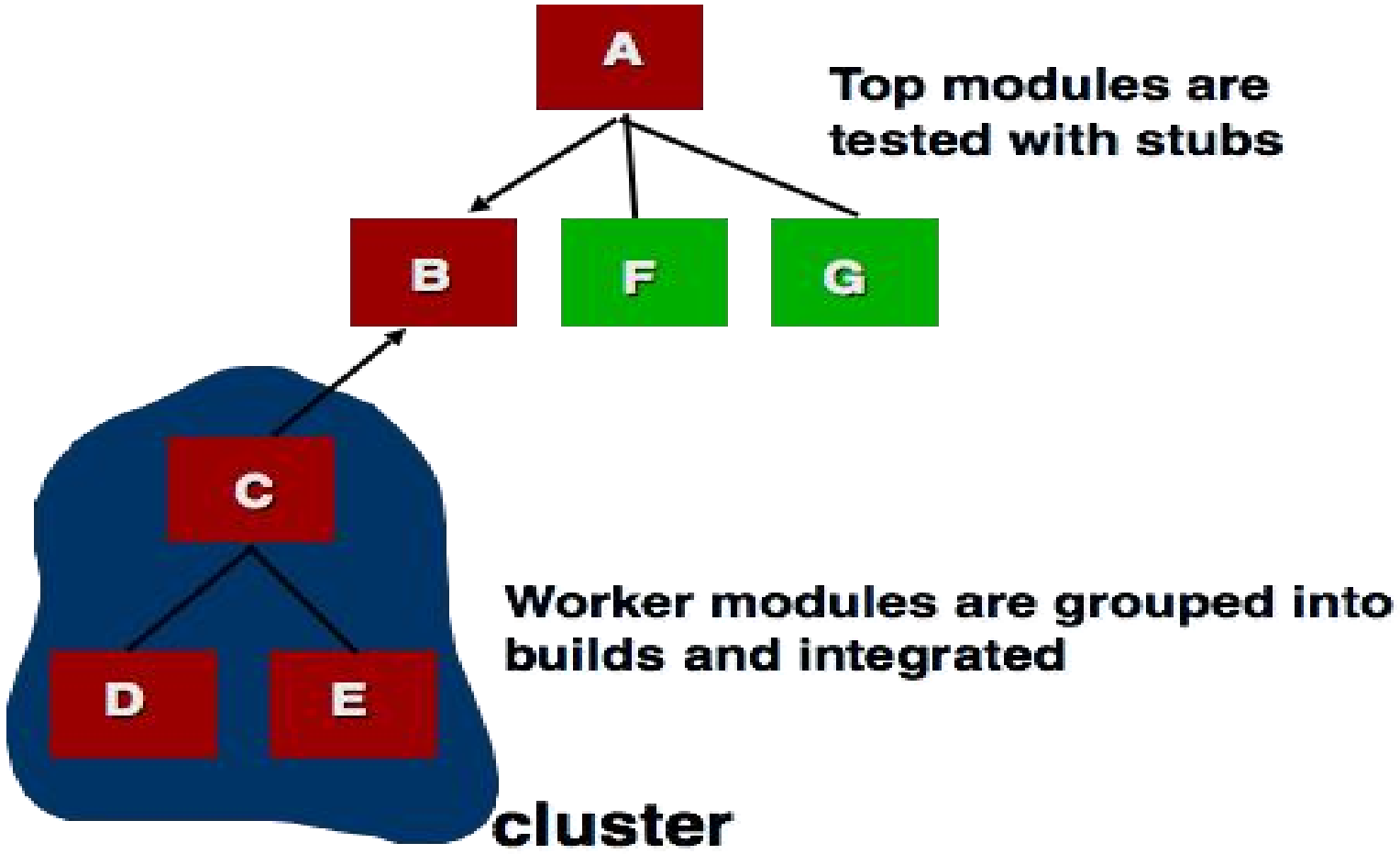


Image source: Pressman, R. (2010). *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., New York, NY, 7th edition

Sandwich Testing

- Process

1. **Identify target layer** based on system characteristics
2. **Use bottom-up** integration **for lower levels** and **top-down** for **higher levels**
3. Continue until all testing **converges** at target layer

- Notes
- Allows integration testing to **begin early**
- **Does not test individual components thoroughly**

Image source: https://commons.wikimedia.org/wiki/File:Peanut-Butter-Jelly-Sandwich.png

Regression Testing

- Regression Test
  - A test applied to a **new version or release** of software to verify that it still **performs the same functions** as the prior version or release
  - Ensures that changes do not have unintended side effects

- Notes
  - The most common approach is to execute all test cases for the system
  - May not be feasible for large systems with lots of test cases

Smoke testing

- **Smoke Test**
- A test applied to a **software build to expose errors** that prevent the build from **performing "core" functionality**
- Designed to **uncover** "show stopper" failures as soon as possible

- A build comprises all data files, libraries, reusable modules, and other components required to implement one or more product functions

A Quote about Debugging:

> Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it? [Kernighan and Plauger, 1978]

Course Module 07: Testing

- **Symptoms and Causes**
  - Symptom and cause may be geographically separated
  - **Symptom**s may. . .
    - .., be **intermittent**
    - .., disappear when another problem is fixed
  - **Causes** may be due to. . .
    - .., a **combination of non-errors**
    - .., a system or compiler error
    - .., assumptions that everyone believes

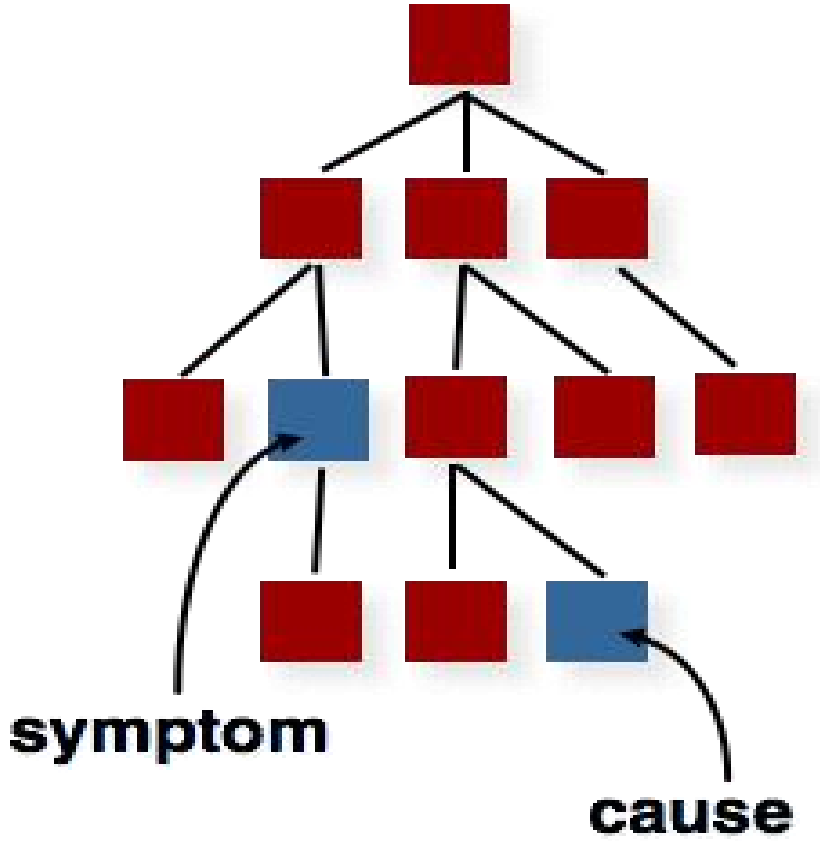


Image source: Pressman, R. (2010). *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., New York, NY, 7th edition

- Consequences of Bugs

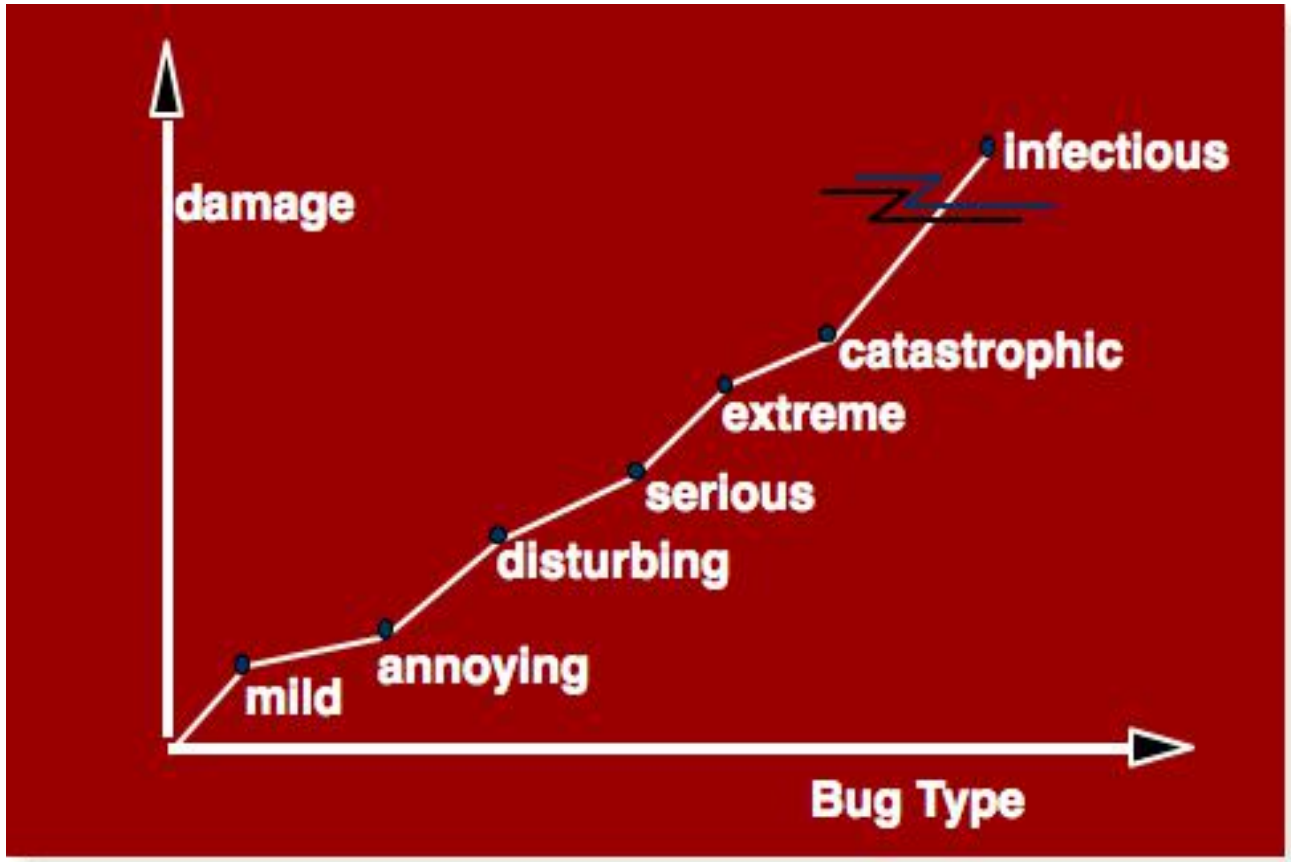


Image source: Pressman, R. (2010). *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., New York, NY, 7th edition

- **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.

Course Module 07: Testing

- Correcting the Error: Notes
  - **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

Course Module 07: Testing

- **Test Case Design**
  - particular choice of input data used when testing a program

Objective	to uncover errors
Criteria	in a complete manner
Constraint	with a minimum of time and effort

Course Module 07: Testing

- Two Approaches
  - **Black box**: View test object as opaque without any implementation details
    - Also known as closed box
  - **White box**: Implementation details about test object known
    - Also known as open box or clear box

Question:

- Which is better—black or white box testing?
- Both approaches are useful, but white box testing allows better coverage of statements and control paths

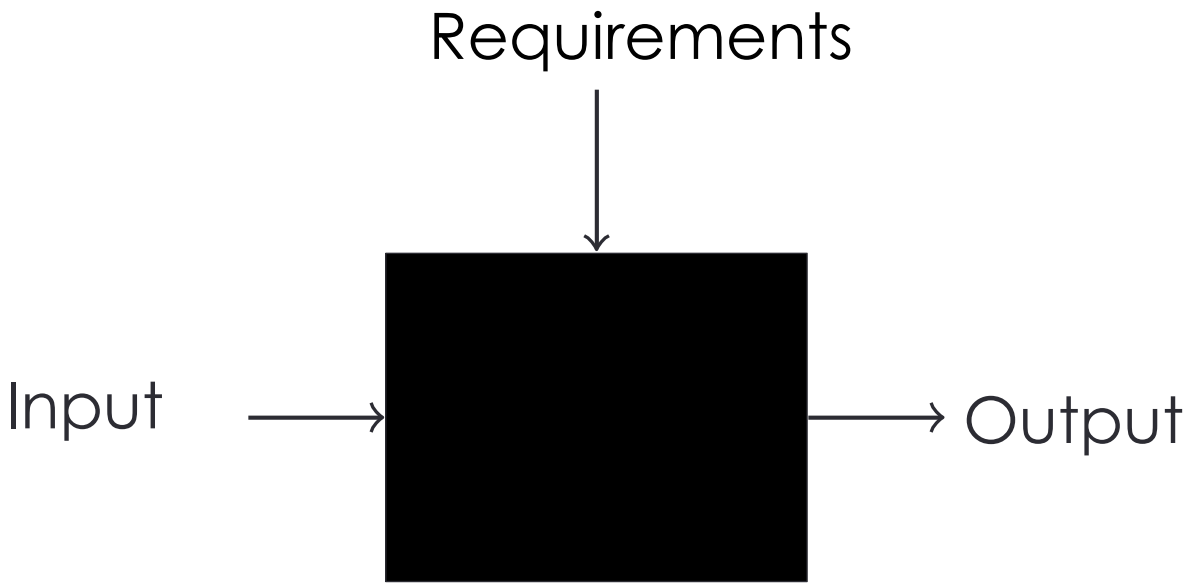
Course Module 07: Testing

In Practice. . .

- . . . it's good to combine black box and white box testing
- Use black box approach for initial test cases
- Use white box approach for additional test cases

Course Module 07: Testing

- Black Box Testing

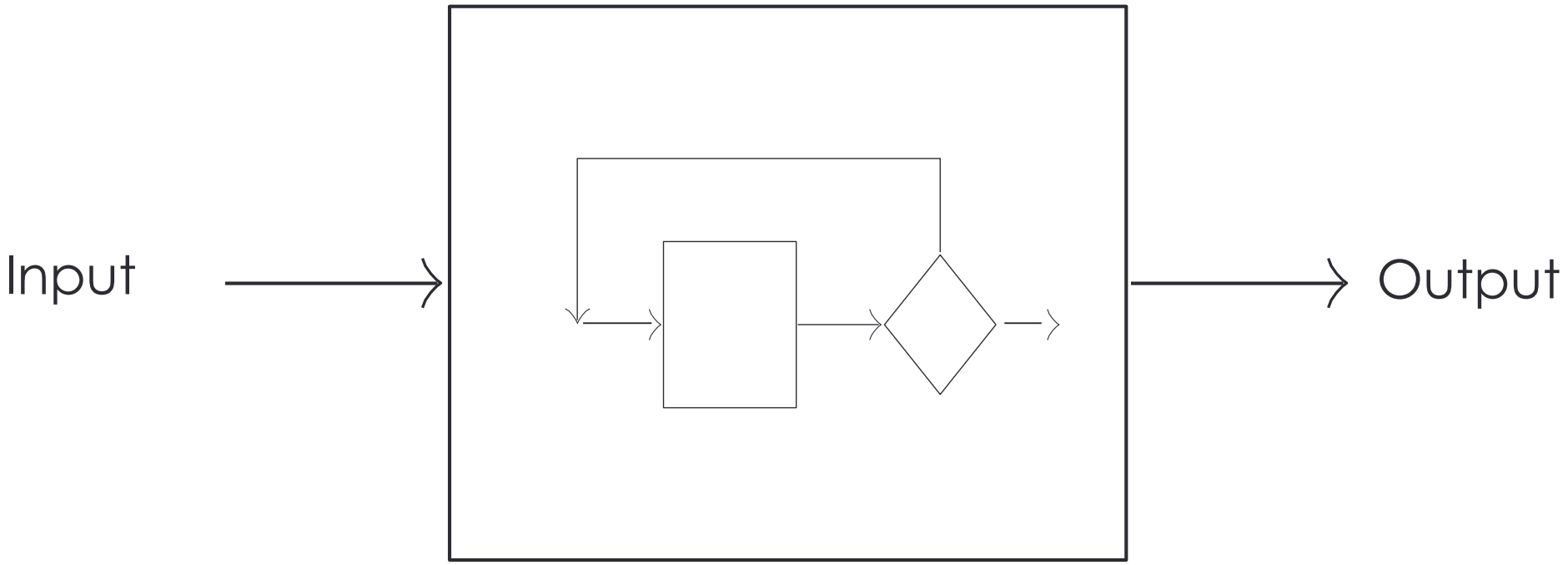


- Black Box Testing Considers:
  - How is functional validity tested?
  - How is system behavior and performance tested?
  - What classes of input will make good test cases?
  - Is the system particularly sensitive to certain input values?
  - How are the boundaries of a data class isolated?
  - What data rates and data volume can the system tolerate?
  - What effect will specific combinations of data have on system operation?

Comparison Testing (Voting Mechanisms)

- Used only in situations in which the **reliability of software is absolutely critical**
- Separate software engineering teams develop independent versions of an application using the same specification
- Each version can be tested with the same test data to ensure that all provide identical output
- All versions are executed in parallel with real-time comparison of results to ensure consistency

- White Box Testing



- Why Cover?
  - Logic errors and incorrect assumptions are inversely proportional to a path's execution probability
  - We often believe that a path is not likely to be executed; in fact, reality is often counter-intuitive
  - Typographical errors are random; it's likely that untested paths will contain some