CSCI 5828: Foundations of Software Engineering

Lecture 24 and 25: Testing Programs

Slides created by Pfleeger and Atlee for the SE textbook

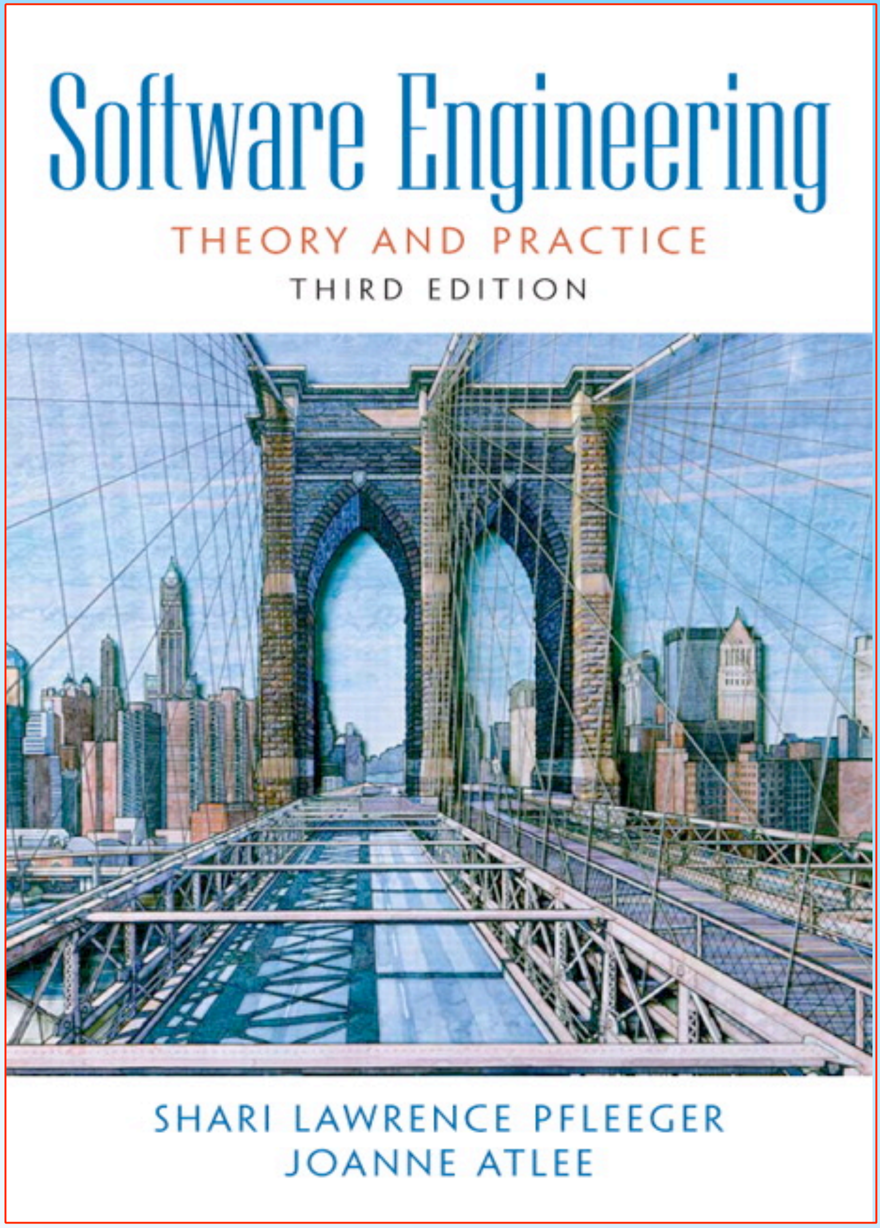
Some modifications to the original slides have been made by Ken Anderson for clarity of presentation

04/10/2008 — 04/15/2008

Chapter 8

Testing the Programs

ISBN 0-13-146913-4
Prentice-Hall, 2006



8.1 Software Faults and Failures
Why Does Software Fail?

- Wrong requirement: not what the customer wants
  – Note: what customer's want changes/evolves over time
- Missing requirement
- Requirement impossible to implement
- Faulty design
- Faulty code
- Improperly implemented design

8.1 Software Faults and Failures
Objective of Testing

- Objective of testing: discover faults
  - Recall: “error, **fault**, failure” framework
  - We are trying to find faults by causing failures to occur
  - We then trace back from the failure to find the fault
- A test is successful only when a fault is discovered
  - **Fault identification** is the process of determining what fault caused a failure
  - **Fault correction** is the process of making changes to the system so that the faults are removed

8.1 Software Faults and Failures
Types of Faults

- Algorithmic fault
- Computation and precision fault
  - a formula's implementation is wrong
- Documentation fault
  - Documentation doesn't match what program does
- Capacity or boundary faults
  - System's performance not acceptable when certain limits are reached
- Timing or coordination faults
- Performance faults
  - System does not perform at the speed prescribed
- Standard and procedure faults

Documentation Fault

- If you find a documentation fault
  - “program behavior does not match documentation”
- which do you change?
  - The program?
  - The documentation?
- What do you think typically happens?
- What should happen?
  - How do you determine what the correct behavior is?

Program Verification

- Program Verification is the **process of demonstrating** that a particular program **meets its specification**
  - If a program meets its specification it is considered “correct”

Program Correctness

- To repeat: **a program is correct only when it meets (i.e. implements) its specification**
- Is the program useful? Not necessarily
  - In order to be useful, the spec. has to match the needs of the system's users
- What happens if the spec. contains an error?
  - If the program matches the spec, its "correct" but the program is not meeting the

Testing Terminology (Review)

- **Error** - a mistake made by a programmer
  - implies that for some input i, F(i) ≠ expected output
- **Fault** - an incorrect state of a program that is entered because of the error
  - **Some errors don’t cause failures right away**, every state between the error and the failure are faults
  - For this class, however, you can think of a “fault” as being the **location in the code where the error exists**
- **Failure** - a symptom of an error
  - e.g. a crash, incorrect output, incorrect behavior, …

Testing Terminology, cont.

- Discussion
  - A **failure occurs only if a fault occurs**, and a **fault occurs only if an error exists**
  - Note: not all faults are detected
    - because you may need to execute a specific portion (e.g. state) of the program for the failure to appear…
    - …and it may be impossible to execute all “states” of a program
  - Recall that Fred Brooks in No Silver Bullet talked about complexity and one aspect of complexity is the sheer number of states associated with software systems

An Example

- If a program contains an error, it does not necessarily lead to a failure

```
if (x < y) /* should be <= */
        …
else
        …
```

- The error may be a typo, or the error could be the result of the programmer not understanding the problem
- The fault is the location of the error, e.g. the expression contained in the if statement, or more explicitly the missing “=”
- A failure may occur if x==y and this if statement is executed

Creating Test Cases

- How do you pick test cases?
  - We will look at two strategies for doing this
    - Black Box Testing (aka Functional Testing)
    - White Box Testing (aka Structural Testing)
  - For now, think of trying to pick “categories” of input that test the same thing

Example

int GreatestCommonDivisor(int x, int y)

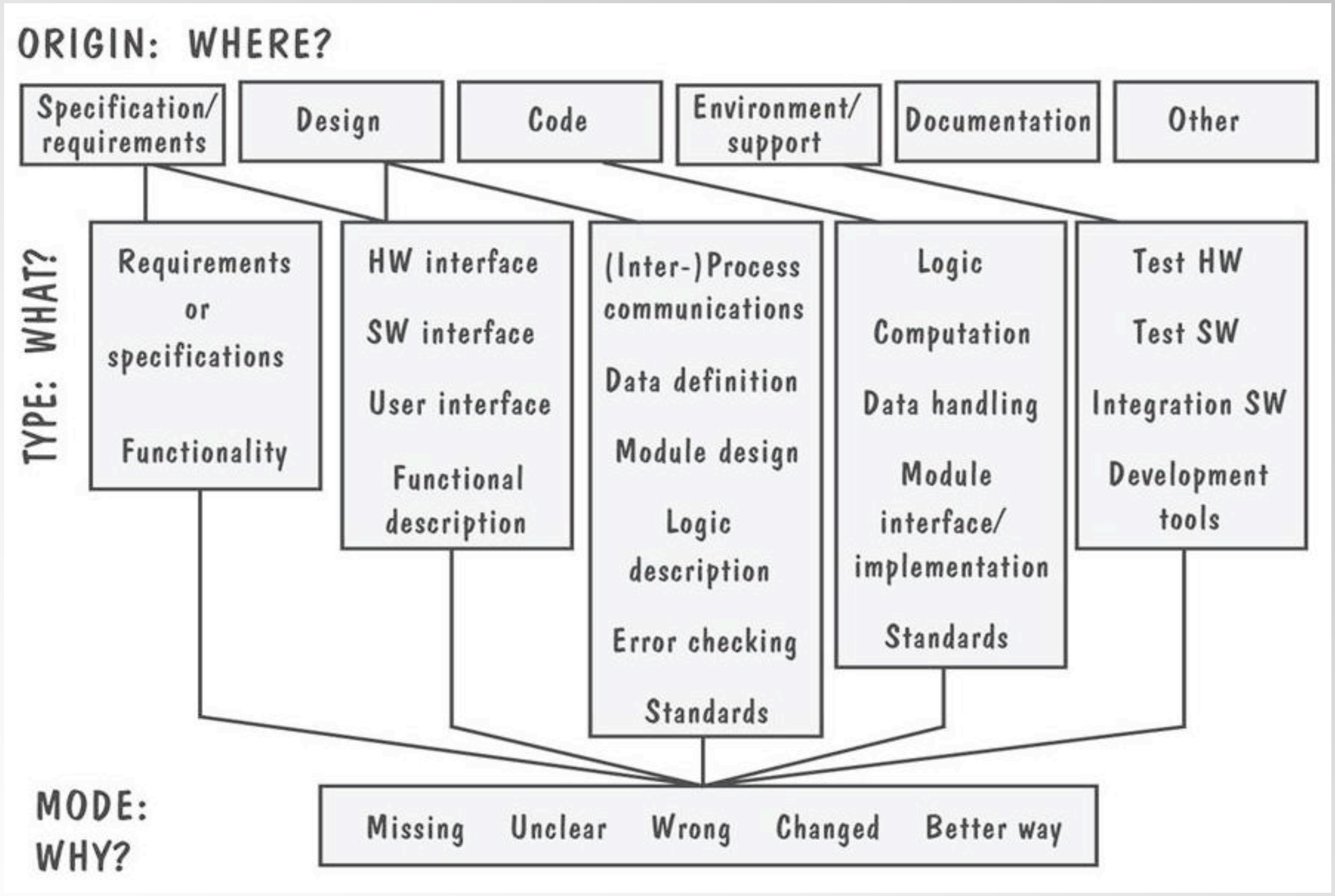
- x=6 y=9, returns 3, tests common case
- x=2 y=4, returns 2, tests when x is the GCD
- x=3 y=5, returns 1, tests two primes
- x=9 y=0, returns ?, tests zero
- x=-3 y=9, returns ?, tests negative

- To test exhaustively is impossible (both parameters can take on an infinite number of values)
  - but with 5 categories identified, we can get by with only 5 test cases!

8.1 Software Faults and Failures
Orthogonal Defect Classification

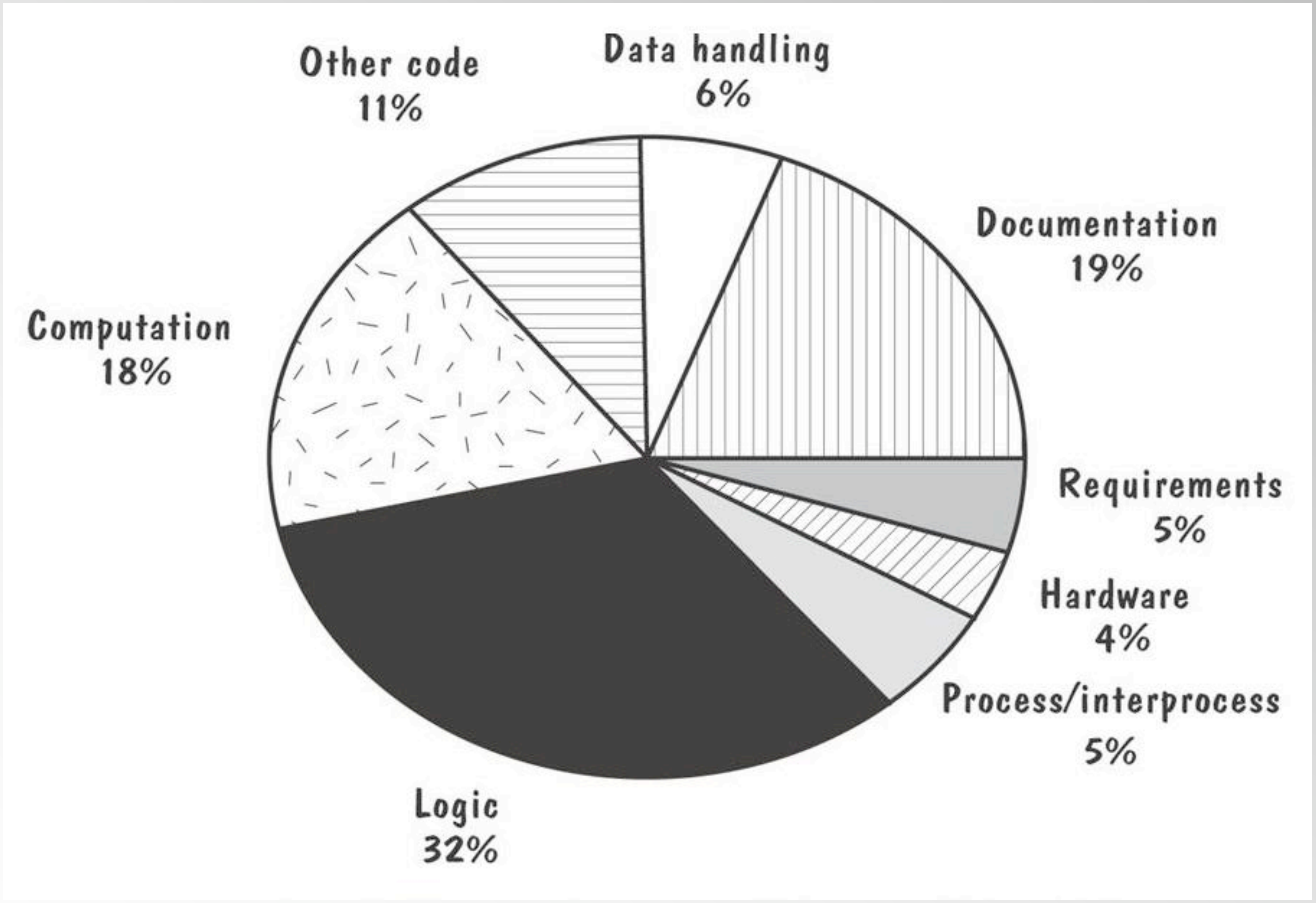
Fault Type	Meaning
Function	Fault that affects capability, end-user interface, product interface with hardware architecture, or global data structure
Interface	Fault in interacting with other component or drivers via calls, macros, control blocks, or parameter lists
Checking	Fault in program logic that fails to validate data and values properly before they are used
Assignment	Fault in data structure or code block initialization
Timing/serialization	Fault in timing of shared and real-time resources
Build/package/merge	Fault that occurs because of problems in repositories, management changes, or version control
Documentation	Fault that affects publications and maintenance notes
Algorithm	Fault involving efficiency or correctness of algorithm or data structure but not design

8.1 Software Faults and Failures
Sidebar 8.1 Hewlett-Packard's Fault Classification



8.1 Software Faults and Failures

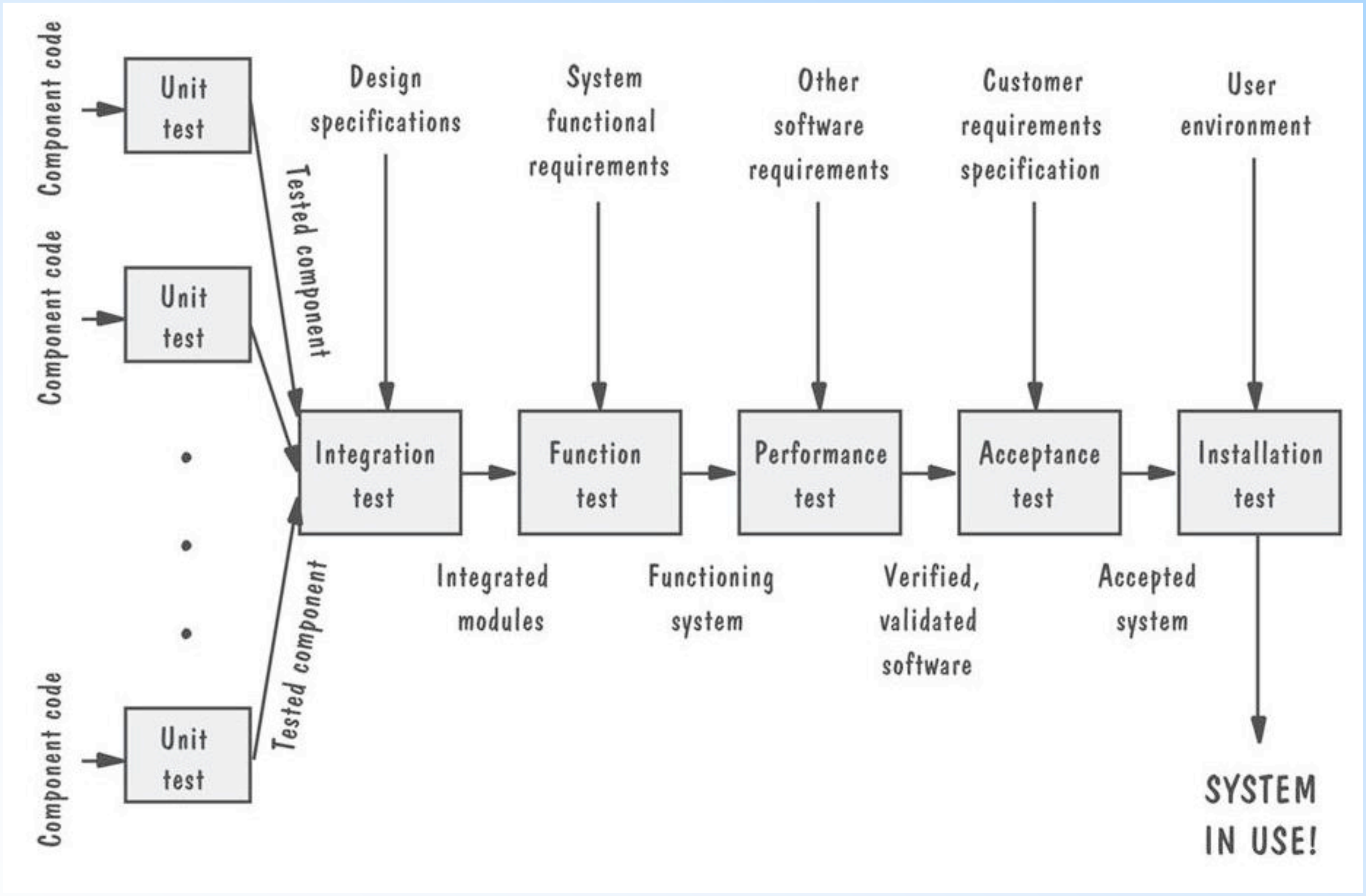
Sidebar 8.1 Faults for one Hewlett-Packard Division



8.2 Testing Issues
Testing Organization

- Module testing, component testing, or unit testing
- Integration testing
- Function testing
- Performance testing
- Acceptance testing
- Installation testing

8.2 Testing Issues
Testing Organization Illustrated



8.2 Testing Issues
Attitude Toward Testing

- Egoless programming: programs are viewed as components of a larger system, not as the property of those who wrote them
  - This can be a hard attitude to adopt for some programmers
  - and is not an easy problem to solve, if it occurs

8.2 Testing Issues
Who Performs the Test?

- Independent test team
  - avoid conflict (within development team)
    - instead create “us vs. them” culture with testing team!
  - improve objectivity
  - allow testing and coding concurrently

8.2 Testing Issues
Views of the Test Objects

- Closed box or black box:
  - functionality of the test objects
- Clear box or white box:
  - structure of the test objects

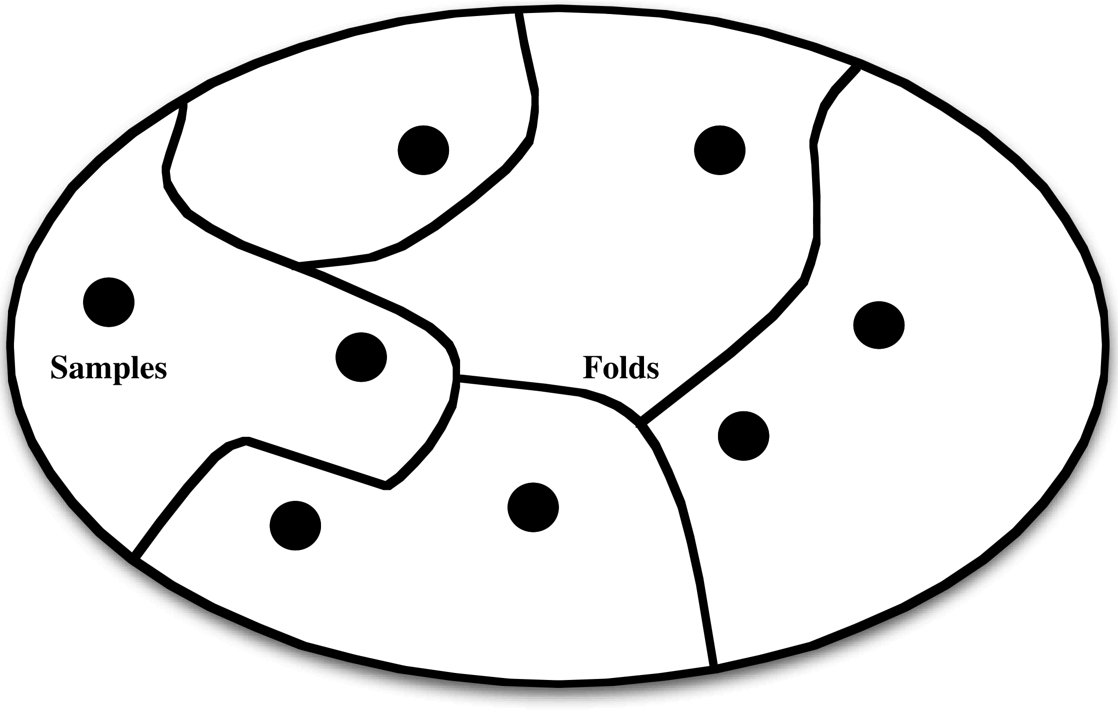
Views of Test Objects

- How do you pick test cases?
- Two main approaches
    - Functional Testing
        - a.k.a. Black Box Testing
    - Structural Testing
        - a.k.a. White Box Testing
- Note: current testing research has moved beyond these concepts…
    - folding and sampling techniques are current
- …but they are used in this class as an introduction

Folding and Sampling

Sample: A specific input to the program

Fold: A class of input that all test the same function



Input Space of Program

Functional Testing

- In functional testing, we test the functionality of the system without regard to its implementation
  - The system is, in a sense, a black box
    - because we cannot look inside to see how it computes its output
  - We provide input and receive output

Functional Testing, continued

- Functional Testing is a strategy for helping a software engineer pick test cases
- This is useful, since selecting test cases is a tricky problem
  - A test suite should be **“complete”**…
    - with respect to the program’s specification
    - but how many test cases do you need to be complete?
  - A test suite should be **precise**
    - no duplicate test cases
    - if a test suite takes too long to run, then it will get run less often (which increases the chance that a fault goes undetected)

Functional Testing, continued

- Functional testing helps create test suites by providing a criterion for selecting test cases:
  - The requirements specification of a program lists functions that the program must perform
  - A test suite is **complete** when it *tests every function*
  - For each function, determine “categories” of input that a function should treat equivalently
    - boundary conditions can be useful guides
    - test both “typical” input and error conditions
    - a test suite will need at least **one test case** for **each category** associated with **each function**

Functional Testing: Step 1

- Identify **functional categories** in the requirements specification that broadly classifies functions the program must perform
  - Example: IT system for Car Dealer
    - Car Database
      - including creation, deletion, and update of entries for each car owned by the dealership
    - Employee Database
    - Payroll
    - Inventory Forecasting
    - Report Generation
    - Sorting

Functional Testing: Step 2

- Identify **specification items** in the spec that correspond to functions the program must perform
- Each specification item should be assignable to one of your functional categories
  - This can be an iterative process, in which a specification item identifies a new functional category
- Example
  - The user shall be able to create a new entry in the car database. A unique id will automatically be assigned to this entry. The user shall then enter additional information about the new car in the fields provided. (Car Database)
  - The user shall be able to see a list of cars sorted by the time a car has been in the inventory from longest to shortest (report generation, sorting)
  - The system shall notify users by e-mail whenever a direct deposit into their bank account has occurred (payroll)

Functional Testing: Step 3

- Identify **functional equivalence classes** for each specification item (like we saw for GCD earlier)
- Example: spec. item
  - The user shall be able to see a list of cars sorted by the time a car has been in the inventory from longest to shortest
- The functional classes might be
  - List is requested on an empty car database
  - List is requested on a car database with a single entry
  - List is requested on a car database with many entries
  - List is requested on a car database that has been edited (entries added, then removed and/or updated)
  - etc. (You need to determine when enough is enough... for some items it may be possible to come up with a ton of categories; restrain yourself and pick only the most critical)

Functional Testing: Step 4

- Determine test cases for each category
    - You may only have one test case per category; however, its okay to pick more than one test case per category
    - Be on the look out for boundary conditions; sometimes they are handled in the categories (i.e. zero, one, or many cars in database), sometimes they are not. In the latter case, you will then need more than one test case in the category to cover each boundary condition
    - This means that in functional testing, “there is more than one way to skin a cat”; one person's categories may be another person's test case, may be another person's specification item. It all depends on the level of granularity that you set for each concept.
- Identifying Test Cases
    - List is requested on an empty car database: one test case (create empty database, invoke function)
    - List is requested on a car database with a single entry: one test case (create empty database, add one car, invoke function)

Functional Testing: Step 5 & 6

- Step 5: Eliminate redundant test cases
  - For example zero cars in the database will probably be a functional equivalence class for several different spec. items;
    - A single test will cover that functional class for all such items
- Step 6: Prioritize test cases
  - You may not have the time or budget to test them all
  - As such, give critical test cases higher priority…
  - …while test cases that test obscure or uncommon errors can be given lower priority
- You now have your test suite!

Structural Testing

- See lecture located here
  - http://www.cs.colorado.edu/~kena/classes/3308/f06/lectures/20/index.html

8.3 Unit Testing
Code Review

- Code walkthrough
  - You present code/documentation you have written to team; focus is on code, not coder!
  - atmosphere is informal; team looks for problems and make’s suggestions
- Code inspection
  - More formal version of walkthrough
  - review team checks code/docs against a prepared list of concerns;
    - initial meeting for overview
    - then careful inspection by each member
    - then second meeting to report findings
- How successful?
  - 82% of faults found in development, found in inspections (Fagan)
  - 93% of faults in 6000 line program found in inspections
  - 85% of fualts in 10M line program found in inspections

8.3 Unit Testing
Fault Discovery Rate

Discovery Activity	Faults Found per Thousand Lines of Code
Requirements review	2.5
Design Review	5.0
Code inspection	10.0
Integration test	3.0
Acceptance test	2.0

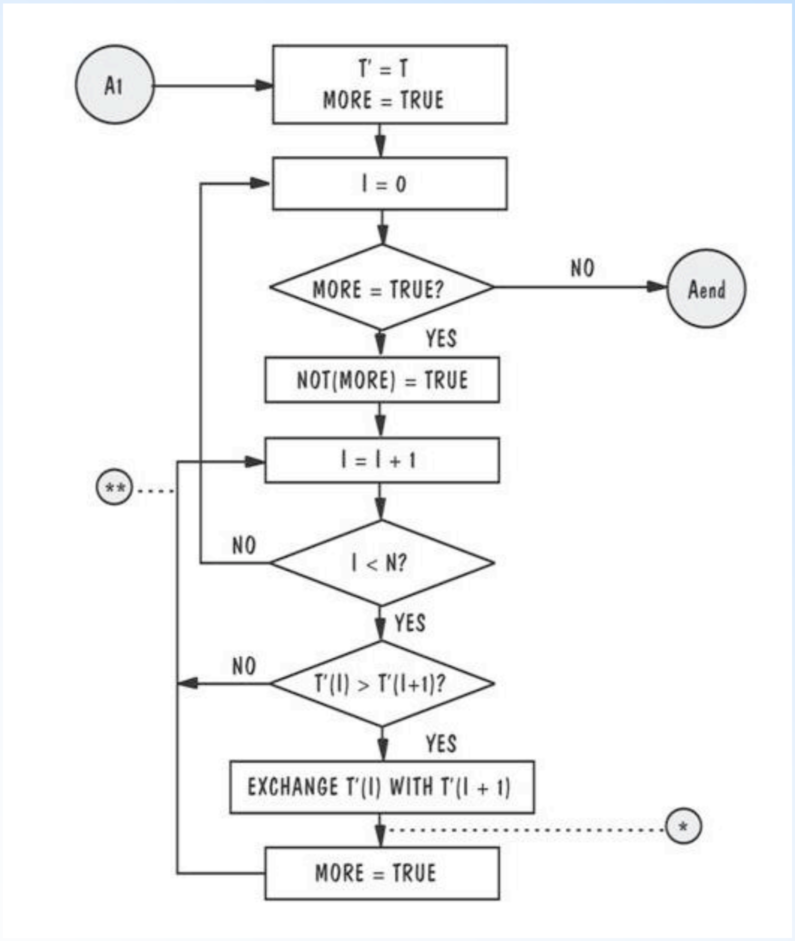
8.3 Unit Testing
Sidebar 8.3 The Best Team Size for Inspections

- The preparation rate, not the team size, determines inspection effectiveness
- The team’s effectiveness and efficiency depend on their familiarity with their product

8.3 Unit Testing
Proving Code Correct

- Formal proof techniques
  - Transform code to logical counterpart
  - Make assertions about input
  - Make assertions about each transformation
  - Make assertions about output
  - Use formal proof techniques (1st-ord. predicate logic)
- Symbolic execution
- Automated theorem-proving

8.3 Unit Testing
Proving Code Correct: An Illustration



Input Assertion:
A1: array(T) & len(T) == N

Output Assertion:

Aend:

array(T') & len(T') == N &
forall(i<N) $\rightarrow$ T'(i) $\leq$ T'(i+1) &
forall(i $\leq$N) $\rightarrow$ exists(j): T'(i) == T(j)

Need to prove that if A1 is true, then Aend is true

Informally:

T' starts out equal to T
We never assign a value to T' that wasn't in T
We always swap higher values with lower values
We do not quit until we go through the entire array without finding a value out of place

8.3 Unit Testing
Testing versus Proving

```
def sort(t):

   assert( type(t) == type([]))

   N = len(t)

   tprime = t[:]
   MORE = True

   while MORE:
       i = 0
       MORE = False
       while i < N - 1:
           if tprime[i] > tprime[i+1]:
               tmp = tprime[i+1]
               tprime[i+1] = tprime[i]
               tprime[i] = tmp
               MORE = True
           i += 1

   assert( type(tprime) == type([]))
   assert( len(tprime) == N)
   for i in range(N-1):
      assert ( tprime[i] <= tprime[i+1] )

   for i in range(N):
      assert ( tprime[i] in t )

   return tprime

print sort([100, 25, 30, 4000, 2, 7, 4, 3500])
```

Output Assertion:

Aend:

array(T') & len(T') == N &
forall($i<N$) → $T'(i) \leq T'(i+1)$ &
forall($i \leq N$) → exists(j): $T'(i) == T(j)$

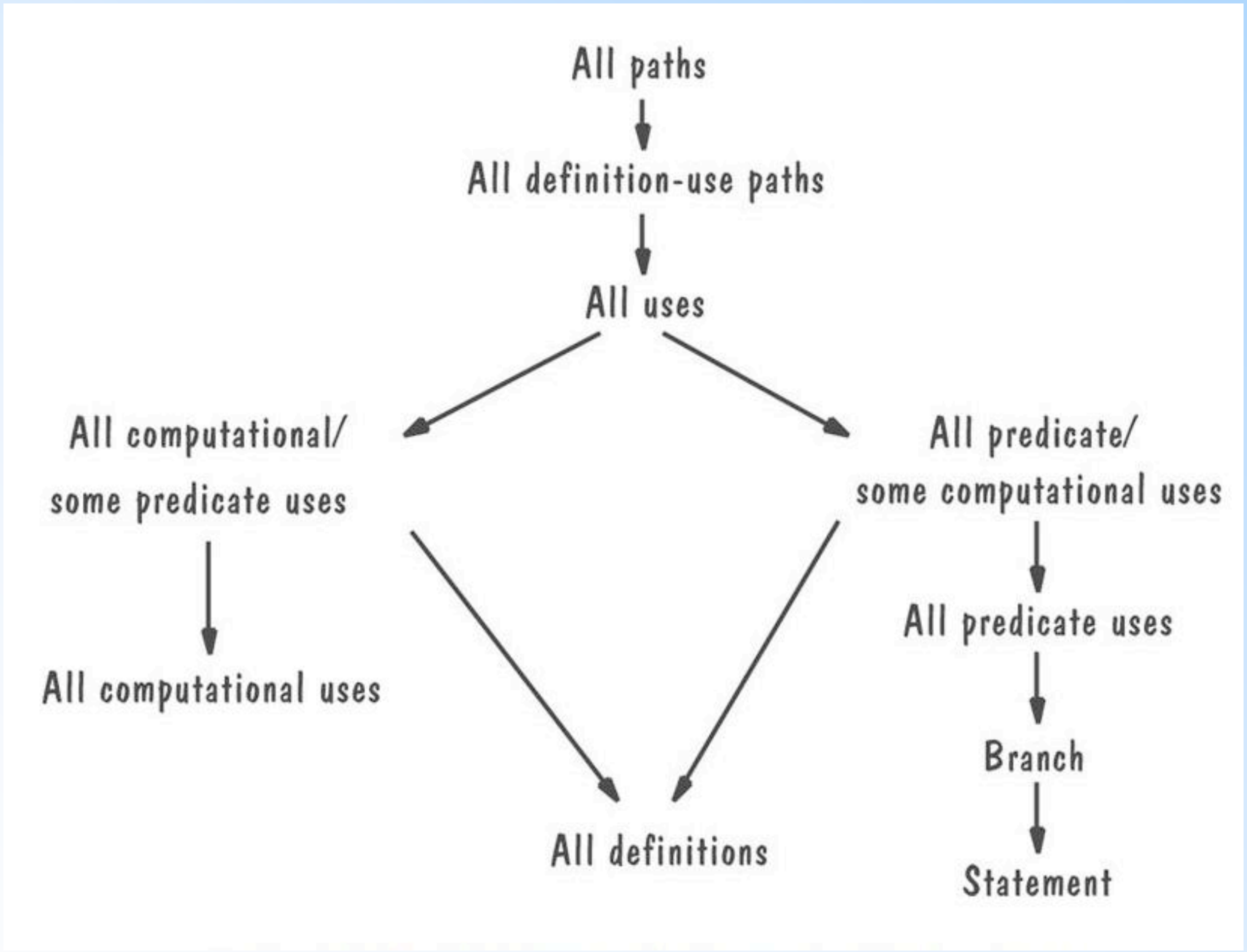
Proofs are hard to do and don't scale to full programs; you can recoup some of the benefits however by using assertions in your code.

At the left, see how the output assertion is translated into six lines of code; you can leave those lines in until you are confident the code is correct; then comment them out before shipping the code

8.3 Unit Testing
Other types of coverage criteria (white box)

- Statement testing
- Branch testing
- Path testing
- Definition-use testing
- All-uses testing
- All-predicate-uses/some-computational-uses testing
- All-computational-uses/some-predicate-uses testing

8.3 Unit Testing
Relative Strengths of Test Strategies



8.3 Unit Testing
Comparing Techniques

- Effectiveness of fault-discovery techniques

	Requirements Faults	Design Faults	Code Faults	Documentation Faults
Reviews	Fair	Excellent	Excellent	Good
Prototypes	Good	Fair	Fair	Not applicable
Testing	Poor	Poor	Good	Fair
Correctness Proofs	Poor	Poor	Fair	Fair

8.3 Unit Testing
Sidebar 8.4 Fault Discovery Efficiency at Contel IPC

- 17.3% during inspections of the system design
- 19.1% during component design inspection
- 15.1% during code inspection
- 29.4% during integration testing
- 16.6% during system and regression testing
- 0.1% after the system was placed in the field

- Demonstrates need for multiple testing techniques in a software life cycle

8.4 Integration Testing

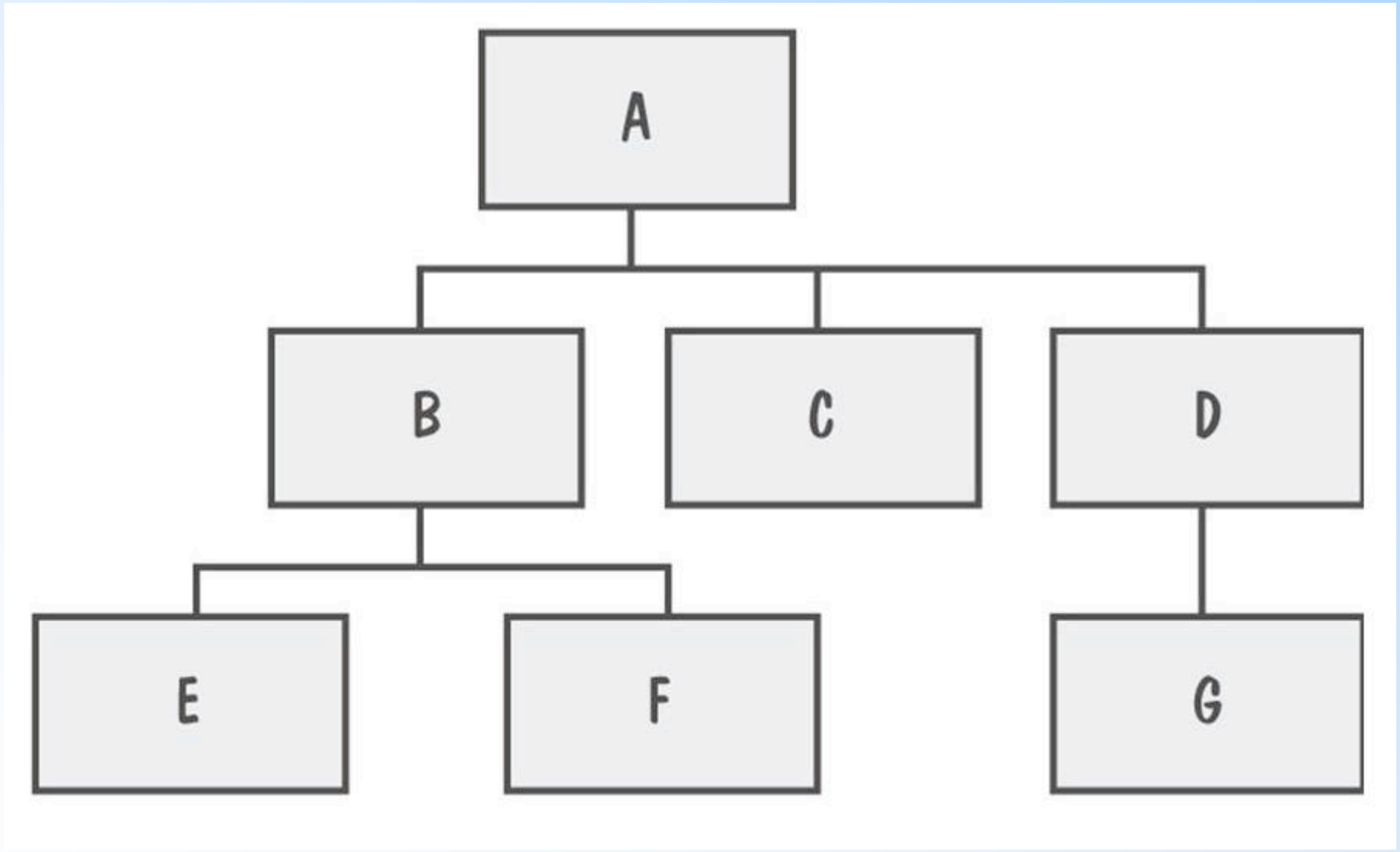
- Bottom-up
- Top-down
- Big-bang
- Sandwich testing
- Modified top-down
- Modified sandwich

8.4 Integration Testing
Terminology

- **Component Driver:** a routine applies a test case on a particular component
- **Stub:** a special-purpose program to simulate the activity of the missing component

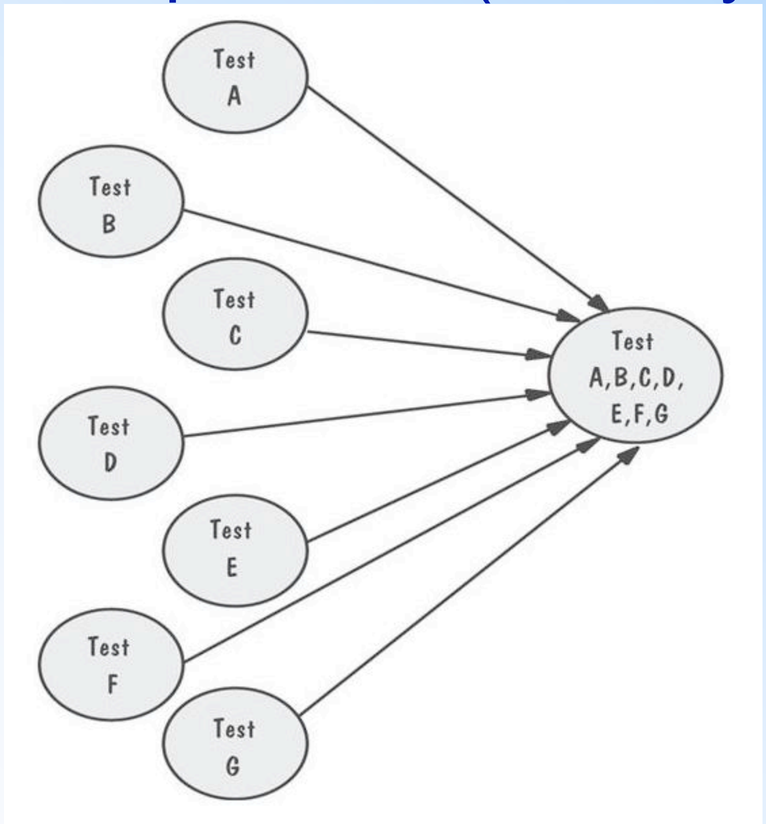
8.4 Integration Testing
View of a System

- System viewed as a hierarchy of components



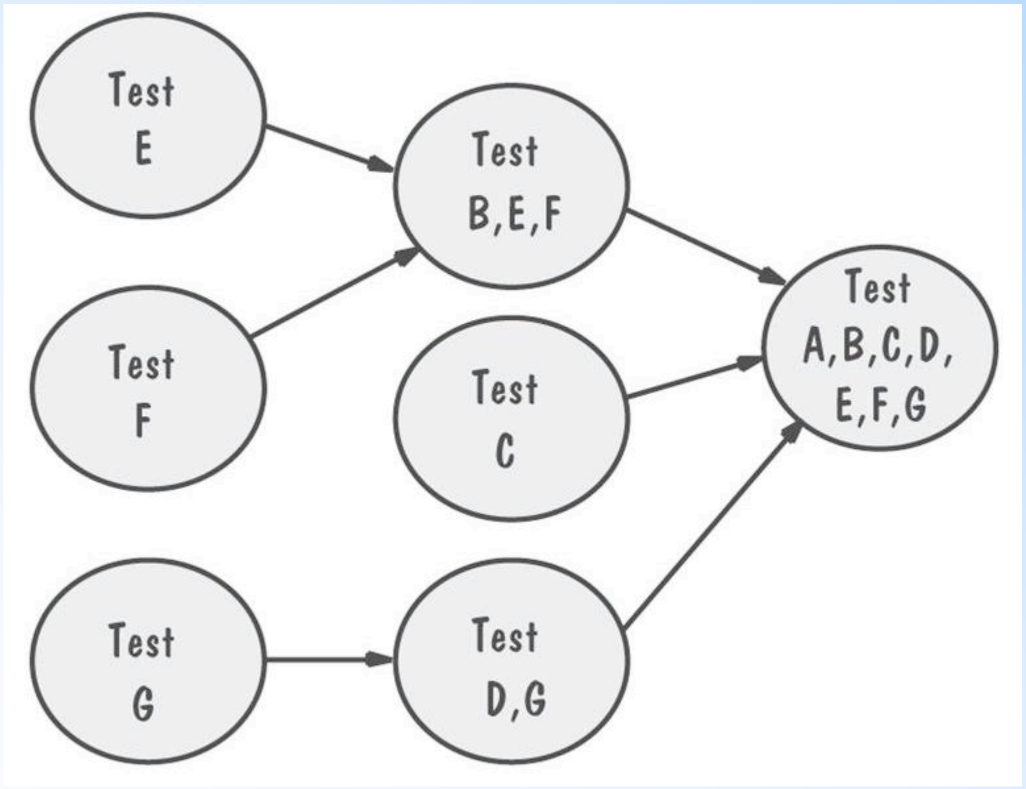
8.4 Integration Testing
Big-Bang Integration Example

- Requires both stubs and drivers to test the independent components; (not very useful)



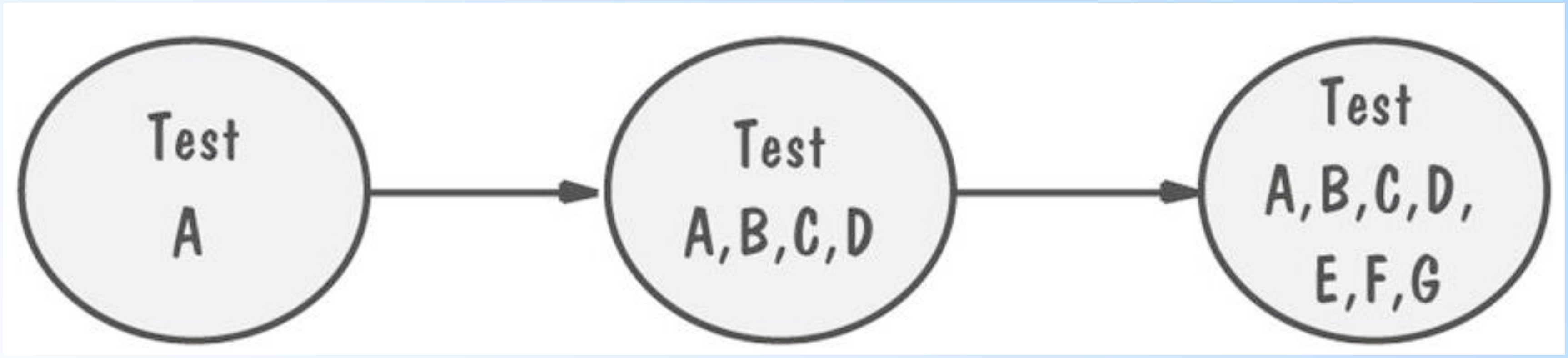
8.4 Integration Testing
Bottom-Up Integration Example

- The sequence of tests and their dependencies



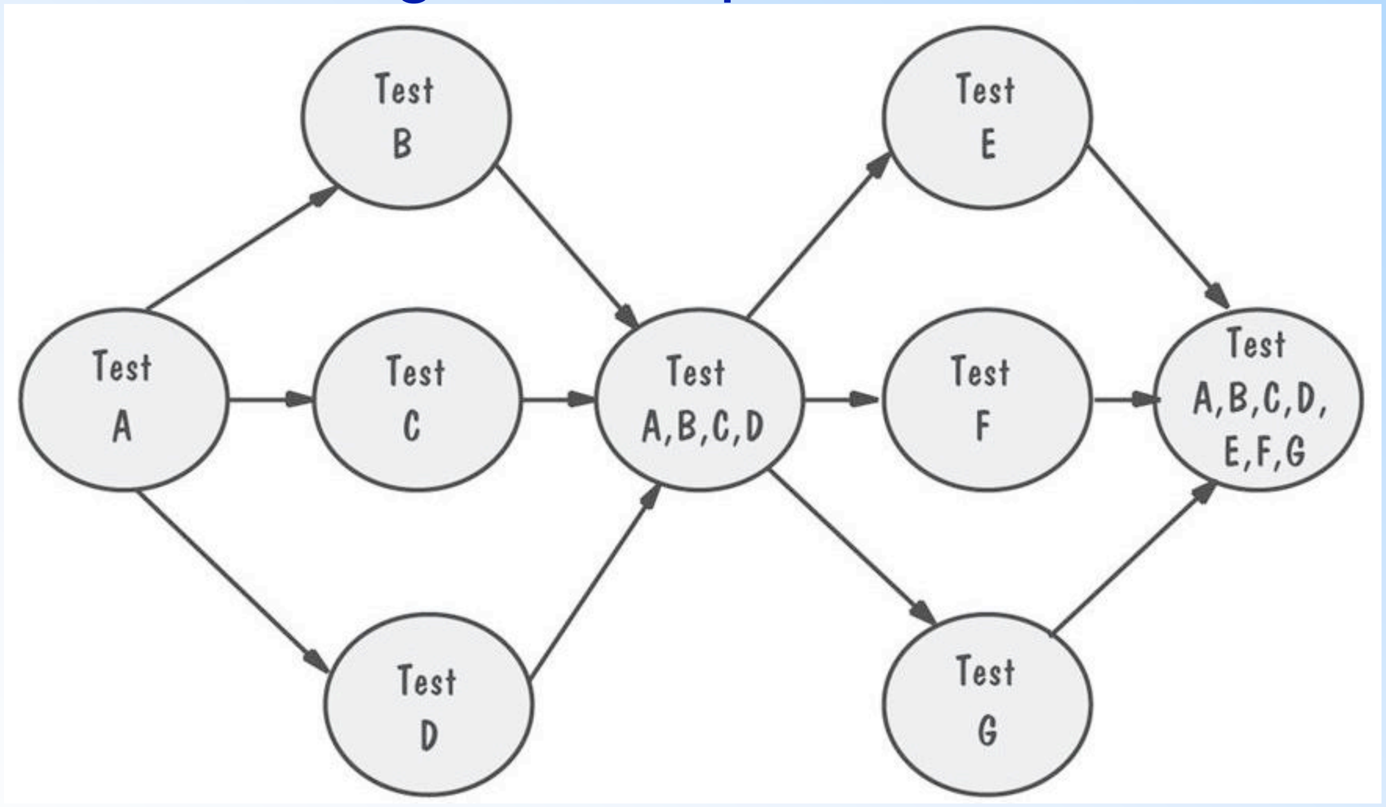
8.4 Integration Testing
Top-Down Integration Example

- Only A is tested by itself
  - Lower level components are simulated via stubs



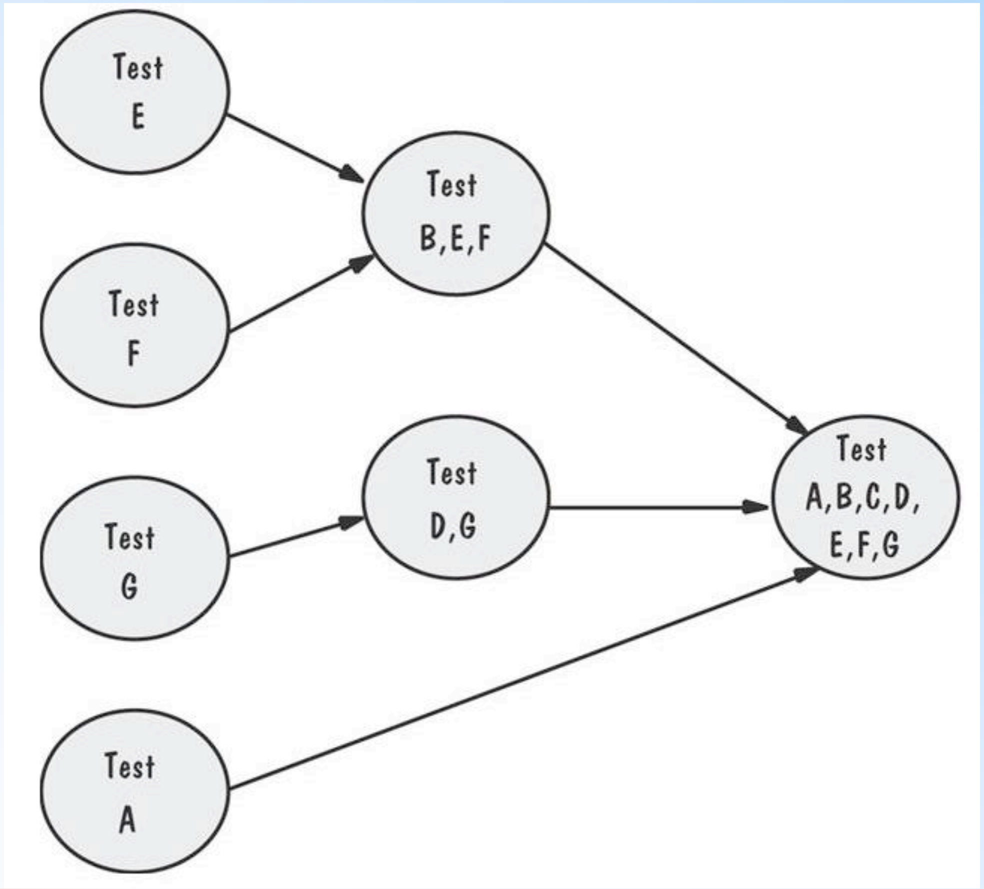
8.4 Integration Testing
Modified Top-Down Integration Example

- Each level's components individually tested before the merger takes place



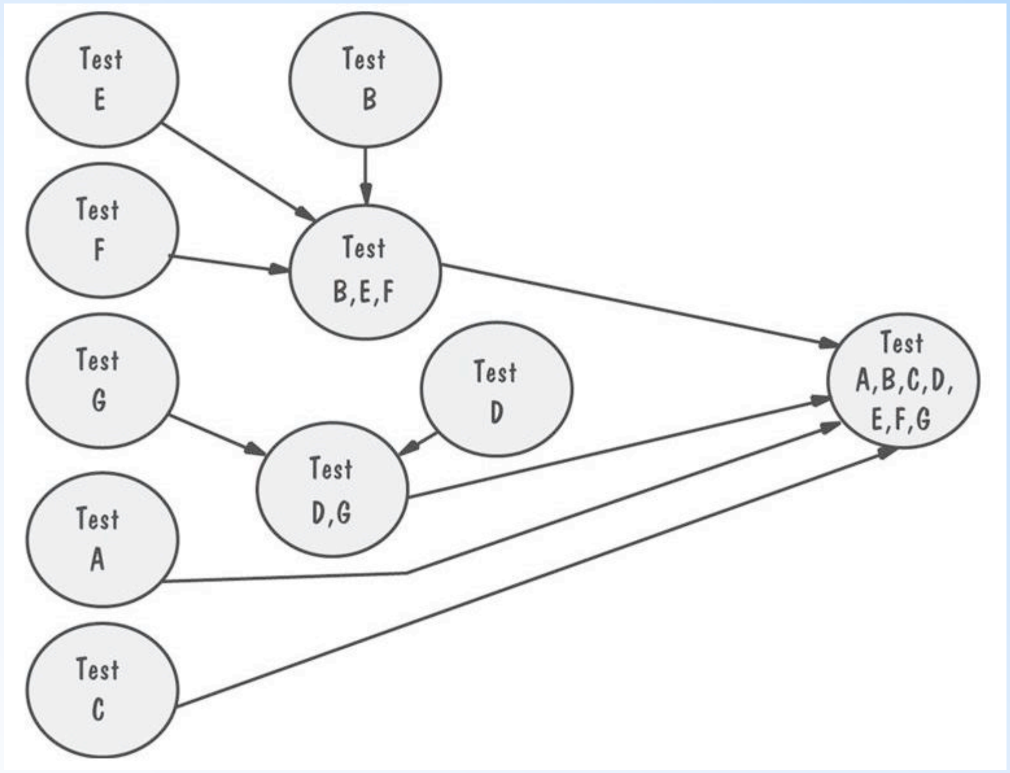
8.4 Integration Testing
Sandwich Integration Example

- Viewed system as three layers



8.4 Integration Testing
Modified Sandwich Integration Example

- Allows upper-level components to be tested before merging them with others

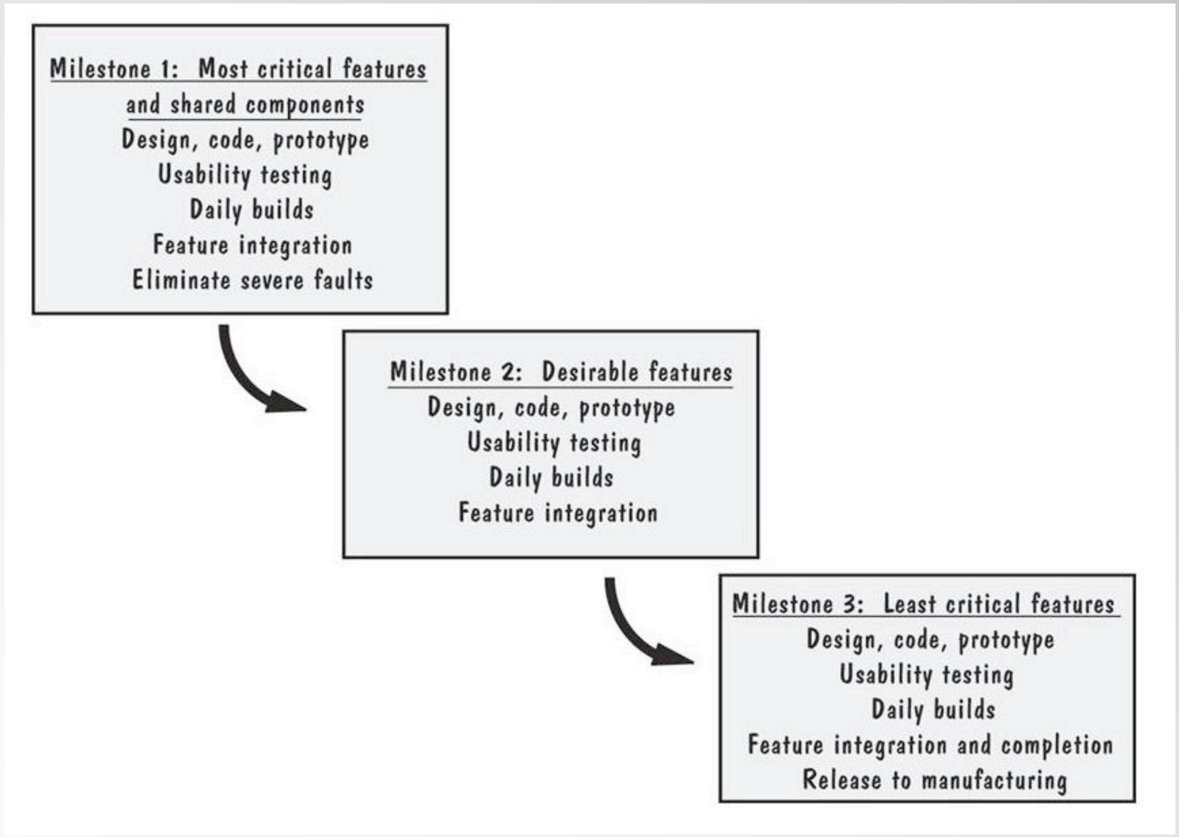


8.4 Integration Testing
Comparison of Integration Strategies

	Bottom-up	Top-down	Modified top-down	Big-bang	Sandwich	Modified sandwich
Integration	Early	Early	Early	Late	Early	Early
Time to basic working program	Late	Early	Early	Late	Early	Early
Component drivers needed	Yes	No	Yes	Yes	Yes	Yes
Stubs needed	No	Yes	Yes	Yes	Yes	Yes
Work parallelism at beginning	Medium	Low	Medium	High	Medium	High
Ability to test particular paths	Easy	Hard	Easy	Easy	Medium	Easy
Ability to plan and control sequence	Easy	Hard	Hard	Easy	Hard	hard

8.4 Integration Testing
Sidebar 8.5 Builds at Microsoft

- The feature teams synchronize their work by building the product and finding and fixing faults on a daily basis



8.6 Test Planning

- Establish test objectives
- Design test cases
- Write test cases
- Test test cases
- Execute tests
- Evaluate test results

8.6 Test Planning
Purpose of the Plan

- Test plan explains
  - who does the testing
  - why the tests are performed
  - how tests are conducted
  - when the tests are scheduled

8.6 Test Planning
Contents of the Plan

- What the test objectives are
- How the tests will be run
- What criteria will be used to determine when the testing is complete

8.11 What this Chapter Means for You

- It is important to understand the difference between faults and failures
- The goal of testing is to find faults, not to prove correctness
- Can use:
  - Functional Testing (black box)
  - Structural Testing (white box)
  - Folding and Sampling
- Proving code correct is difficult and non-scalable; assertions can lower costs and achieve similar benefits
- After unit testing, integration testing begins
- Testing for large systems requires a test plan