CS342 Software Engineering

Dr. Ismail Hababeh
German Jordanian University

Lecture 18

TESTING WORKFLOW

Adapted from Software Engineering, by Dr. Paul E. Young
& slides by Dr. Mohammad Daoud

Post-Delivery Maintenance

- More cost is spent on post-delivery maintenance than other workflows
- Two types of testing are needed during this phase
  - Testing the changes made during post-delivery maintenance
  - Regression testing (verifies that the software that was previously developed and **tested** still performs correctly after it was changed or interfaced with other software)

Planning for Testing

- The SPMP (Software Project Management Plan) must explicitly state what testing should be done
- Test cases must be drawn up as soon as possible after the specifications are completed.

The Test Workflow

- The test workflow is the responsibility of
  - Every developer and maintainer
  - The quality assurance QA group
- Traceability of artifacts is an important requirement for successful testing

Test Cases Types

- Random testing: worst testing way as it may skip testing serious cases.
- Test all cases: it takes too long time.
- A systematic way is needed to conduct a set of all possible cases that covers the product functionalities.

Testing to Specifications vs. Testing to Code

- There are two strategies of testing
- *Test to specifications* (also called black-box, data-driven, functional, or input/output driven testing)
  - Ignore the code, use the specifications to select test cases
- *Test to code* (also called glass-box, logic-driven, structured, or path-oriented testing)
  - Ignore the specifications, use the code to select test cases

Feasibility of Testing to Specifications

- Example 1:
  - The specifications for a data processing product include 5 types of commission and 7 types of discount ⇨ 35 test cases
- We can't say that commission and discount are computed in two entirely separate artifacts. Hence, the code structure is irrelevant.

Feasibility of Testing to Specification

- Example 2:

Suppose a software product specifications include 20 factors, each taking on 4 values

  - There are 4^{20} test cases
  - If each takes 30 seconds to run, then running all test cases takes more than 1 million years

- The exponential computation makes testing to specifications impossible

Feasibility of Testing to Code – Example 1

- Each path through an artifact must be executed at least once.

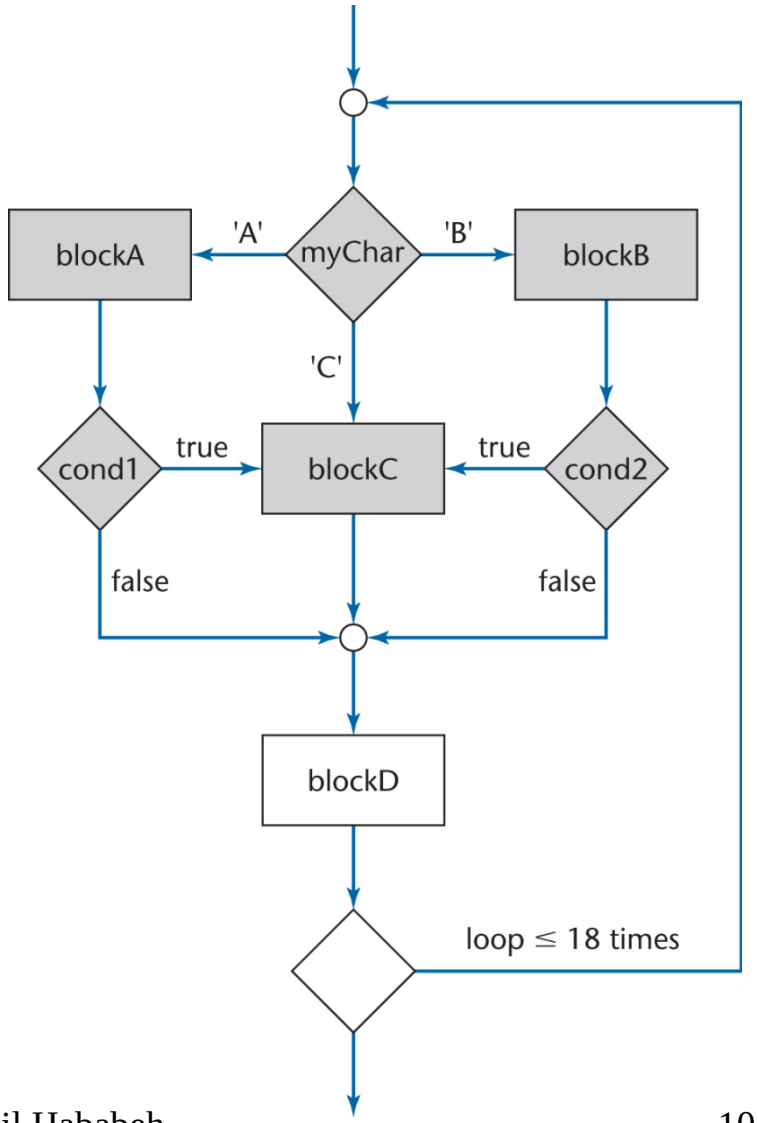
```
read (kmax)                                  // kmax is an integer between 1 and 18
for (k = 0; k < kmax; k++) do
{
   read (myChar)                             // myChar is the character A, B, or C
   switch (myChar)
   {
     case 'A':
       blockA;
       if (cond1) blockC;
       break;
     case 'B':
       blockB;
       if (cond2) blockC;
       break;
     case 'C':
       blockC;
       break;
   }
   blockD;
}
```

Flowchart of Example 1

- The flowchart has 5^{18} different paths:

A,C,D
A,D
B,C,D
B,D
C,D
Total of 5 different options in 18 trials (loop)

5^1 + 5^2 + 5^3 .. 5 ^18



Feasibility of Testing to Code – Example 2

- Testing to code is not reliable

```
if ((x + y + z)/3 == x)
   print "x, y, z are equal in value";
else
   print "x, y, z are unequal";
```

Test case 1: x = 1, y = 2, z = 3
Test case 2: x = y = z = 2

- We can exercise every path without detecting every fault

Feasibility of Testing to Code – Example 3

- A path can be tested only if it is present
- A programmer who omits the test for d = 0 in the code probably is unaware of the possible danger

```
if (d == 0)
   zeroDivisionRoutine ();
else
   x = n/d;
```

(a)

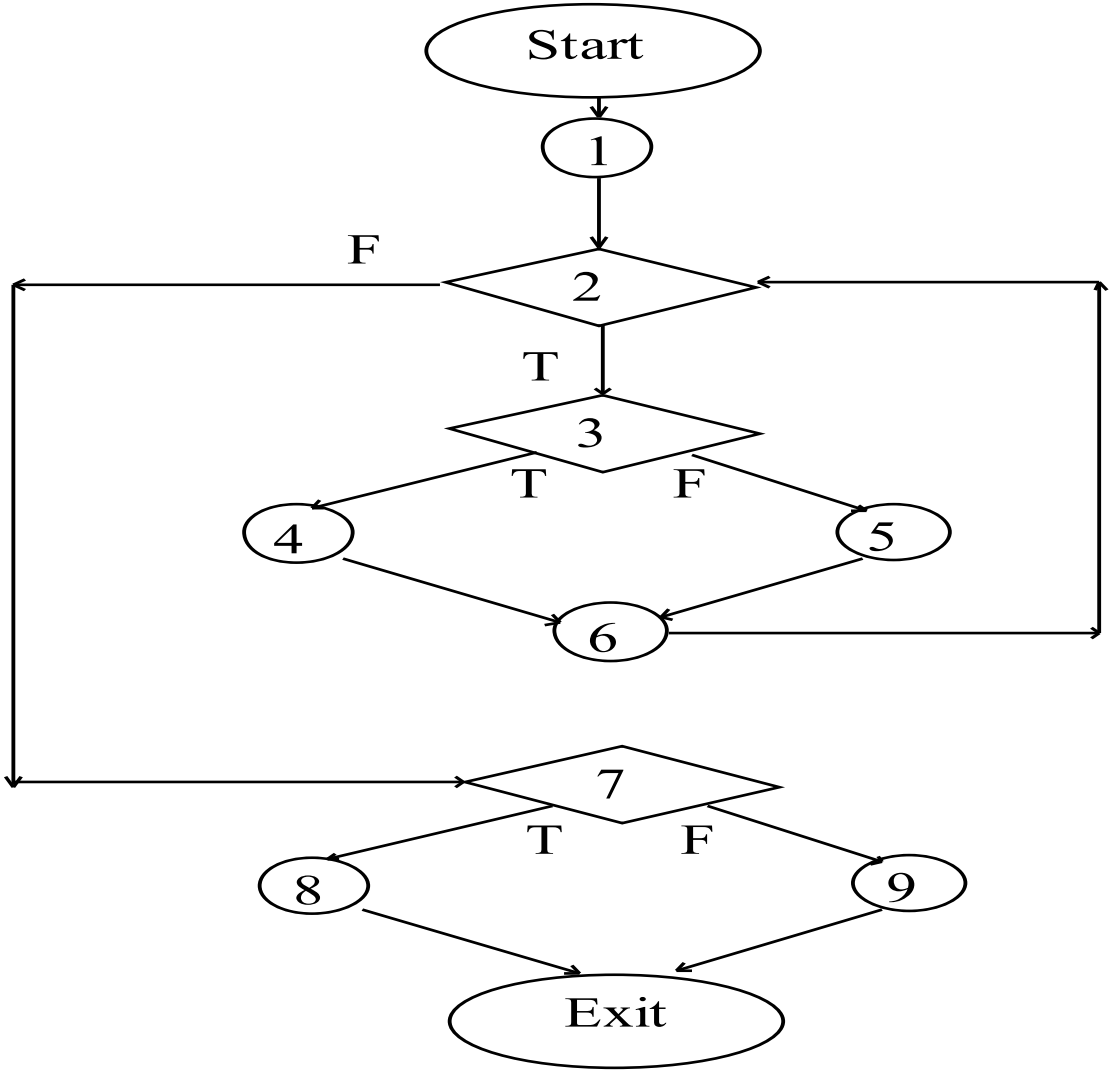
```
x = n/d;
```

(b)

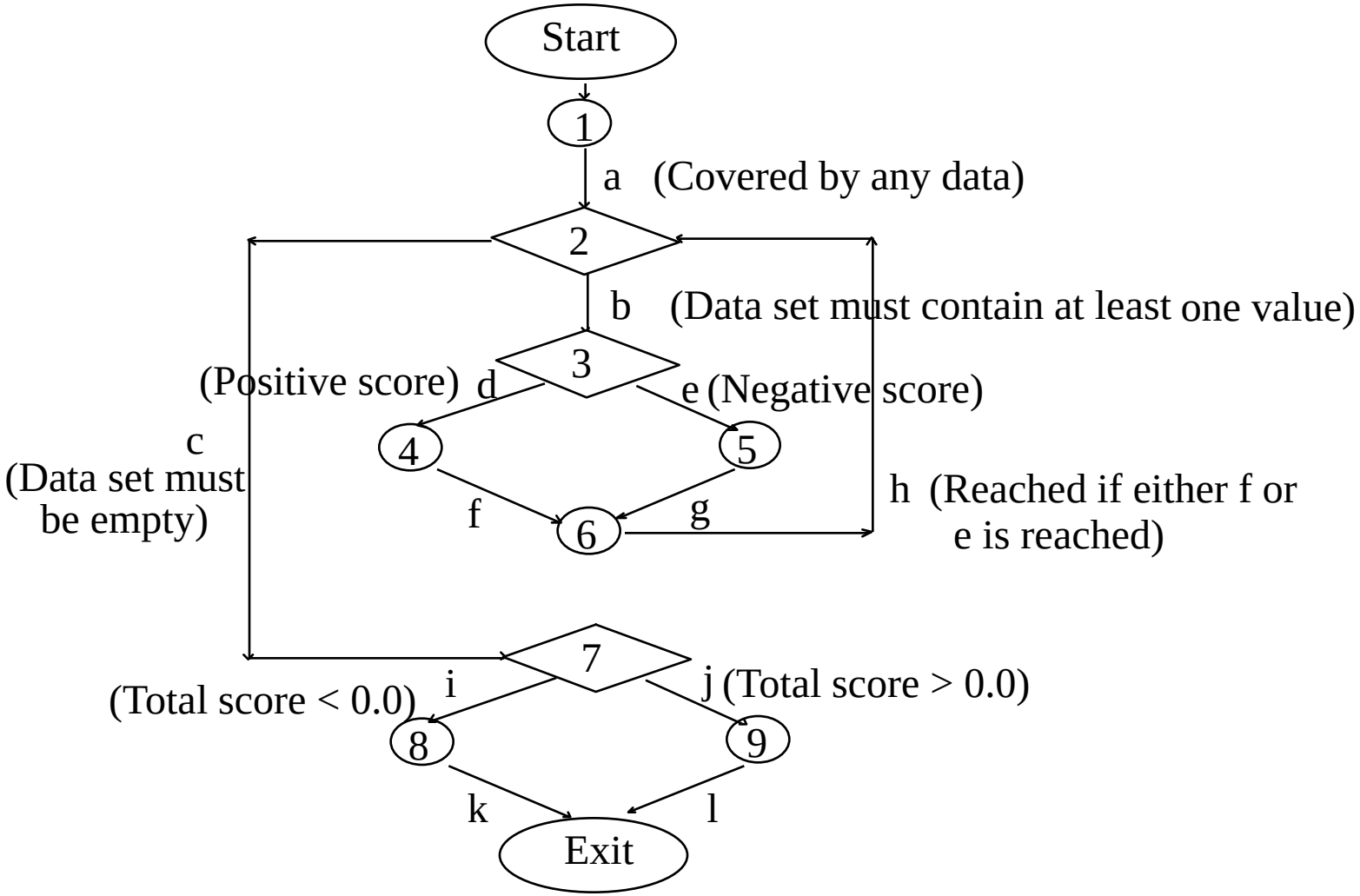
Determining Test Paths – Example 4

```
FindMean (FILE ScoreFile)
{   float SumOfScores = 0.0;                                        (1)
    int NumberOfScores = 0;
    float Mean=0.0; float Score;
    Read(ScoreFile, Score);
(2) while (! EOF(ScoreFile) {
(3)     if (Score  > 0.0 ) {
            SumOfScores = SumOfScores + Score;                      (4)
            NumberOfScores++;
        }                                                           (5)
    Read(ScoreFile, Score);                                         (6)
    }
    /* Compute the mean and print the result */
(7) if (NumberOfScores > 0) {
        Mean = SumOfScores / NumberOfScores;                        (8)
        printf(" The mean score is %f\n", Mean);
    } else
        printf ("No scores found in file\n");                       (9)
}
```

Constructing the Logic Flow Diagram



Finding the Test Cases



Guidance for Test Case Selection

- Use *Analysis Knowledge* about functional requirements (black-box):
  - ✓ Use cases
  - ✓ Expected input data
  - ✓ Invalid input data
- Use *Design Knowledge* about system control structures, data structures (white-box):
  - ✓ Control structures
    - ➢ Test branches, loops, ...
  - ✓ Data structures
    - ➢ Test record fields, arrays, ...
- Use *Implementation Knowledge* about algorithms:
  - ✓ Force division by zero
  - ✓ Use sequence of test cases for interrupt handler

Testing Terminology

- Reliability: denotes the software trustworthiness and dependability, it can be determined by measuring the probability of the product working correctly over a given period.
- Fault (Bug): A software procedure or algorithm cause of an error.
- Error: A software state in which further processing by the product will lead to a failure. Note that not all errors may lead to a failure.
- Failure: Any deviation of the product behavior

Testing Strategies

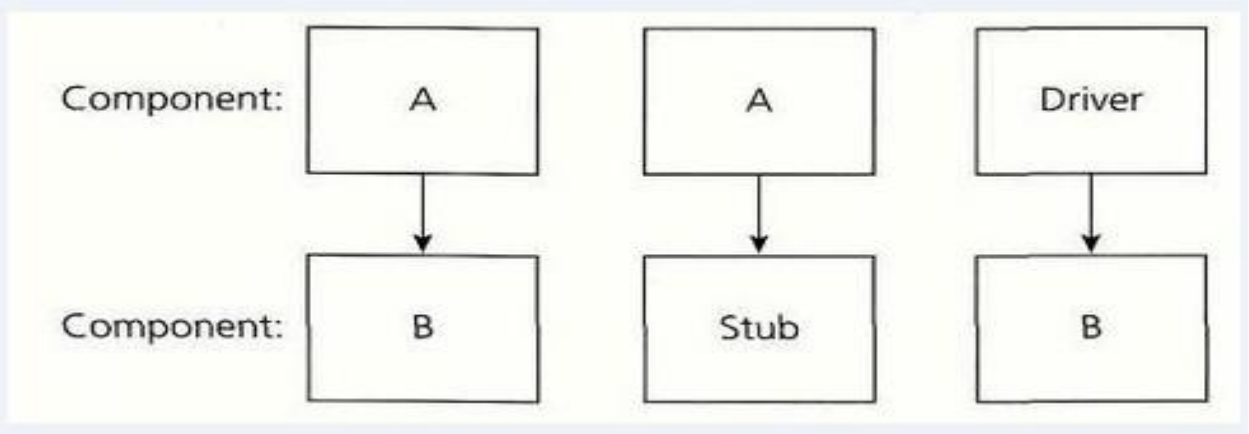
- Unit testing
  - Test each artifact separately
- Integration testing
  - Component testing
  - Stubs and Drivers
  - Top-Down integration
  - Bottom-Up integration
- Product testing
  - Test the functionality of the whole product

Stubs and Drivers

- Stubs and Drivers are used to replace the missing software and simulate the interface between the software components in a simple manner.
- The concept of Stubs and Drivers are mostly used in the case of component testing.
- Component testing may be done in isolation with the rest of the system depending upon the context of the development cycle.

Stubs and Drivers - Example

- Suppose you have a function (A) that calculates student's GPA in a particular academic year.
- Suppose function (A) derives its values from function (B) which calculates the grades obtained in particular subjects.



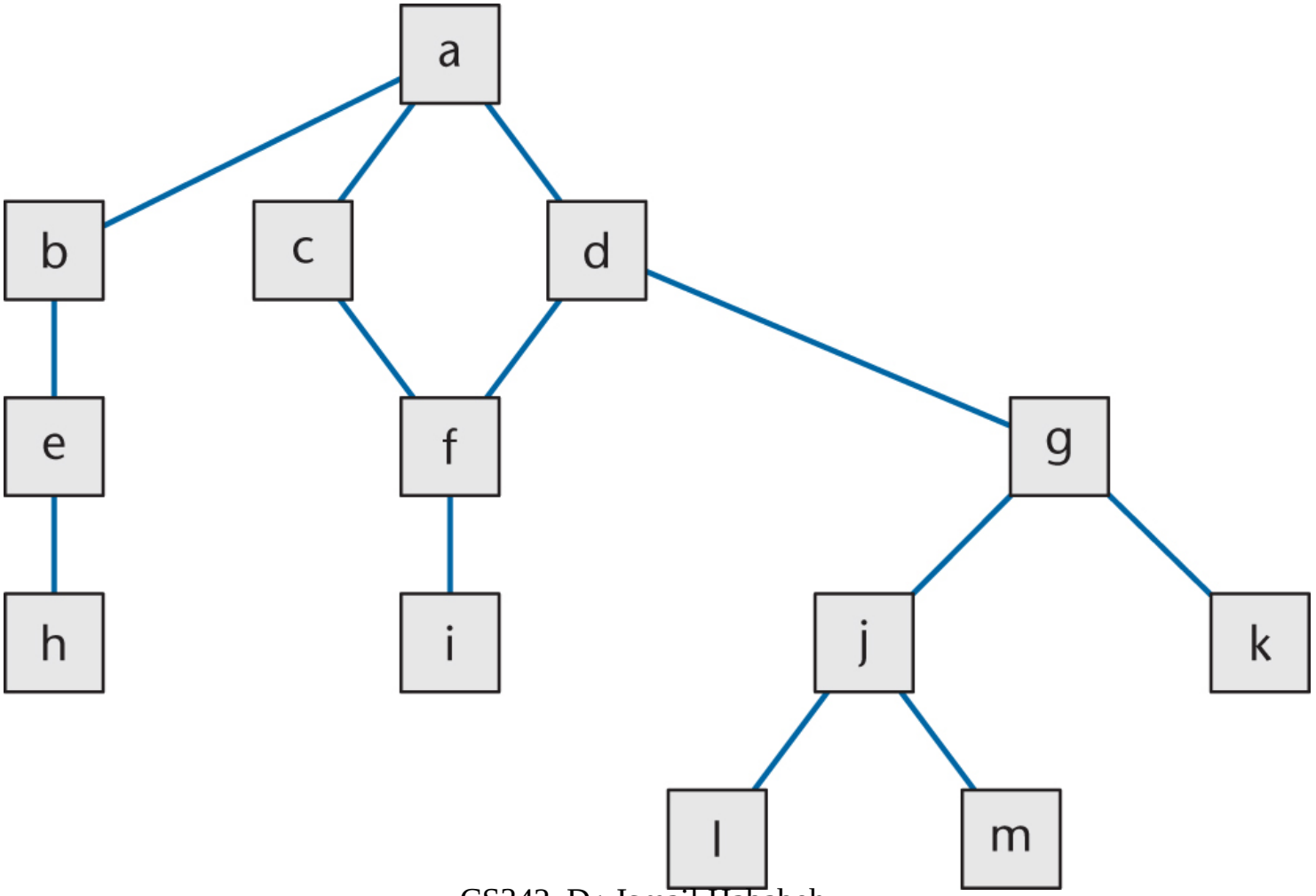
Stubs

- Assume we have finished working on Function A and wants to test it.
- But we can't run the function A without input from function B; Function B is still under development.
- In this case, we create a dummy function to act in place of function B to test function A. This dummy function called (needed) by another function is called a Stub.

Drivers

- Suppose we have finished function B and is waiting for function A to be developed.
- In this case we create a dummy function in place of function A to test function B. This dummy is named Driver.

Integration Testing – Example
A Product with 13 Modules



Implementation and Integration

1. Code and test each artifact separately (Unit testing)
2. Link all 13 artifacts together and test (Integration testing)
3. Test the whole product (Product testing).

Testing Drivers and Stubs - Examples

- To test artifact **a** ⇨ artifacts **b, c, d** must be stubs
- To test artifact **h** on its own ⇨ requires a driver **e**
- Testing artifact **d** ⇨ requires a driver **a** and two stubs: **f,** g.

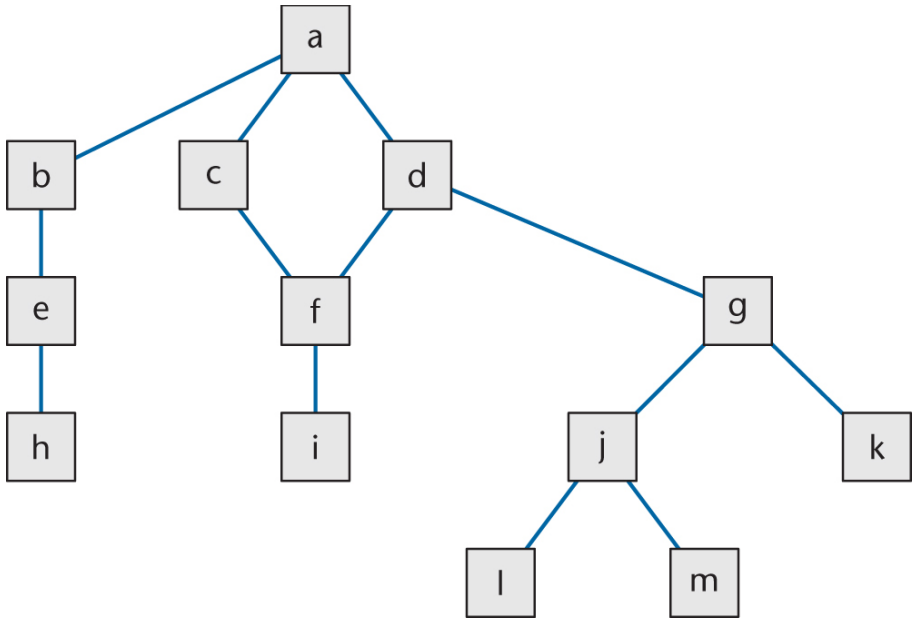
Testing Drivers and Stubs Problems

- Stubs and drivers must be written, then thrown away after unit testing is complete.
- Fault isolation:
  - A fault could be occurred in any of the 13 artifacts or 13 interfaces (in the last example).
  - In a large product with, for example, 100 artifacts and 120 interfaces, there are 220 places where a fault might be occurred.
- Solution ⇨ Combine of unit and integration testing.

Top-down Horizontal Integration Testing

- The code of Top artifact is implemented, integrated and tested before the Down artifact.

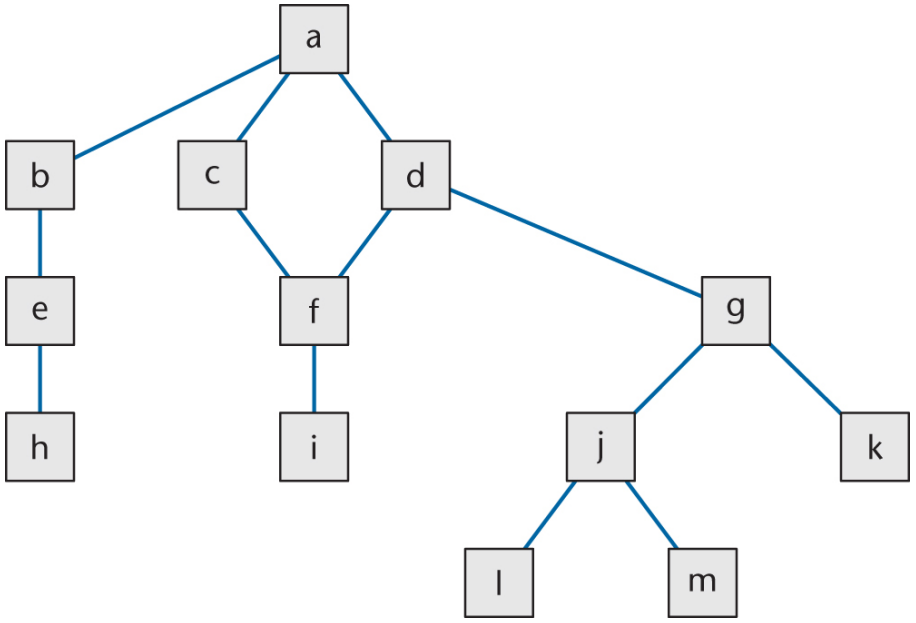
1. One possible of top-down order is (**horizontal testing**).
   - a
   - b, c, d
   - e, f, g
   - h, i, j, k
   - l ,m



Top-down Vertical Integration Testing

2. Another possible top-down ordering is (**vertical testing**).

```
          a
[a]       b, e, h
[a]       c ,d, f, i
[a, d]    g, j, k, l, m
```



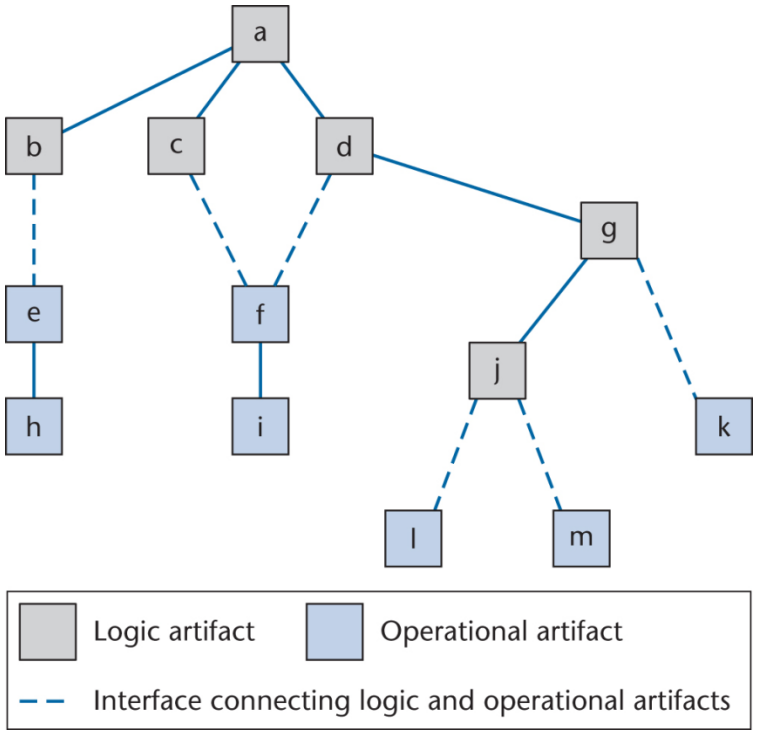
Top-down Integration Testing Advantages

- Fault isolation
  - A previously successful test cases might fail when a new artifact is added to what has been tested so far
    - The fault could occur in the new artifact or the interface(s) between the new artifact and the rest of the product
- Stubs are not wasted
  - Each stub is expanded into the corresponding complete artifact at the appropriate step
- Major design faults show up early

Top-down Logic and Operational Artifacts

- Logic artifacts (Top) include the decision-making flow of control
  - In the example, artifacts `a, b, c, d, g, j`
- Operational artifacts (Down) perform the actual operations of the product
  - In the example, artifacts `e, f, h, i, k, l, m`

➢ The logic artifacts are developed before the operational artifacts



Top-down Integration Testing Problems

- Reusable artifacts are not properly tested
- Lower level (operational) artifacts are not tested frequently
- Defensive programming (fault shielding)

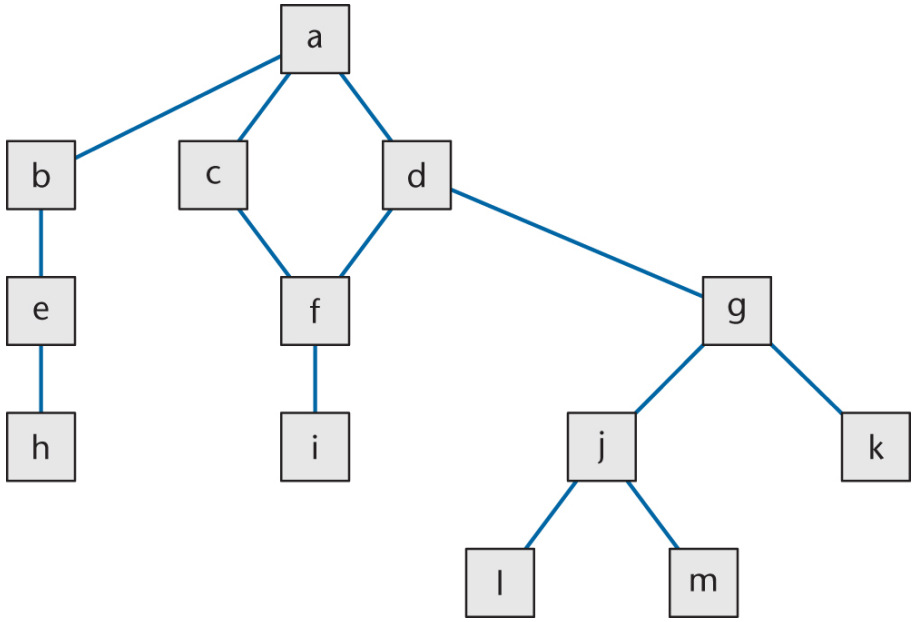
  Example:

```
if (x >= 0)
 y = computeSquareRoot(x, errorFlag);
```

  - `computeSquareRoot` is never tested with `x < 0`
  - This has implications for reuse

Bottom-up Horizontal Integration Testing

- The code of Down artifact is implemented, integrated and tested before the Top artifact.



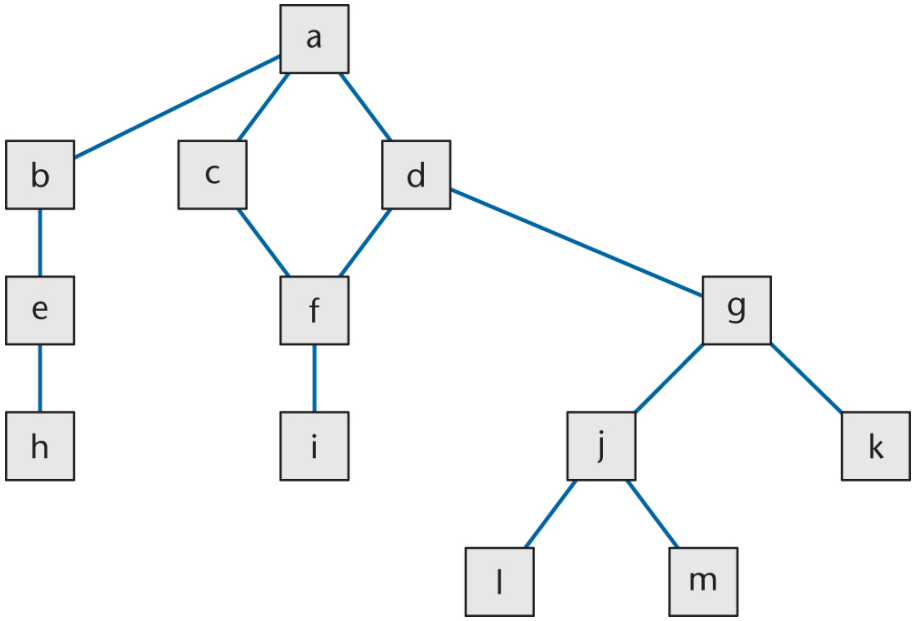
1. One possible bottom-up order is (**horizontal testing**):

```
l, m, h, i, j, k,
  e,f, g, b, c,
  d, a
```

Bottom-up Vertical Integration Testing

2. Another possible bottom-up ordering is (**vertical testing**).

```
h, e, b
i, f, c, d
l, m, j, k, g[d]
a    [b, c, d]
```



`a` is tested when `b, c, d` are developed and tested.

Testing Bottom-up Integration Advantages

- Operational artifacts are thoroughly tested
- Operational artifacts are tested with drivers, not by fault shielding, defensively programmed artifacts.*
- Fault isolation

* Defensive programming is a form of defensive design intended to ensure the continuing function of a piece of software under unforeseen circumstances. Defensive programming practices are often used where high availability, safety, or security is needed.

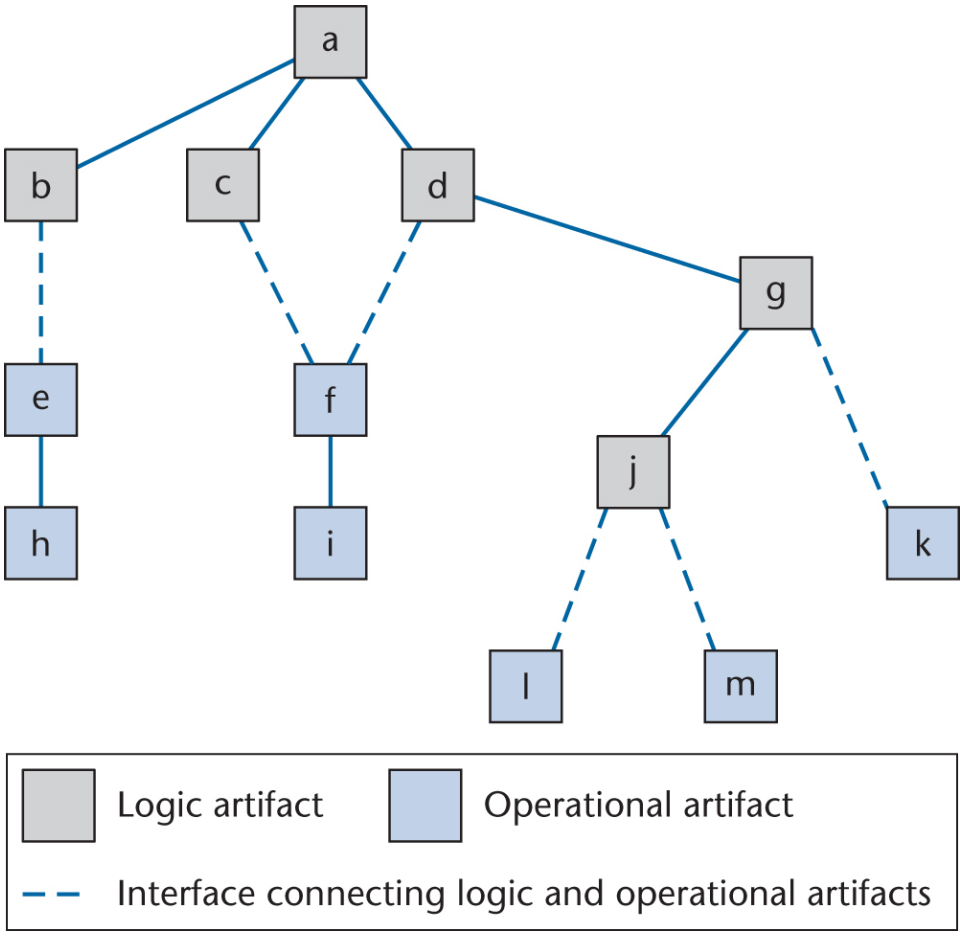
* Wikipedia

Bottom-up Integration Testing Problems and Solutions

- Problem
  - Major design faults are detected late
- Solution
  - Combine top-down and bottom-up strategies making use of their strengths and minimizing their weaknesses

Combine Testing Integration Types

1. Logic artifacts are integrated top-down
2. Operational artifacts are integrated bottom-up
3. Finally, the interfaces between the two groups are tested



Combine Integration Testing Advantages

- Major design faults are caught early
- Operational artifacts are thoroughly tested
- They may be reused with confidence
- Fault isolation always exists.

Integration Testing of Object-Oriented Products

- Object-oriented implementation and integration
  - Combine implementation and integration
  - Objects are integrated bottom-up
  - Other artifacts are integrated top-down