# IMPLEMENTATION

Adapted from Schach

# Overview

- Choice of programming language
- Fourth generation languages
- Good programming practice
- Coding standards
- Code reuse
- Code walkthroughs and inspections

# Choice of Programming Language (contd)

- Example
  - QQQ Corporation has been writing COBOL programs for over 25 years
  - Over 200 software staff, all with COBOL expertise
  - What is “the most suitable” programming language?

- Obviously COBOL

# Choice of Programming Language (contd)

- What happens when new language (C++, say) is introduced
  - C++ professionals must be hired
  - Existing COBOL professionals must be retrained
  - Future products are written in C++
  - Existing COBOL products must be maintained
  - There are two classes of programmers
    - COBOL maintainers (despised)
    - C++ developers (paid more)
  - Expensive software, and the hardware to run it, are needed
  - 100s of person-years of expertise with COBOL are wasted

# Choice of Programming Language (contd)

- The only possible conclusion
  - COBOL is the “most suitable” programming language
- And yet, the “most suitable” language for the latest project *may* be C++
  - COBOL is suitable for only data processing applications
- How to choose a programming language
  - Cost–benefit analysis
  - Compute costs and benefits of all relevant languages

# Choice of Programming Language (contd)

- Which is the most appropriate object-oriented language?
  - C++ is (unfortunately) C-like
  - Thus, every classical C program is automatically a C++ program
  - Java enforces the object-oriented paradigm
  - Training in the object-oriented paradigm is essential before adopting any object-oriented language
- What about choosing a fourth generation language (4GL)?

# 15.2 Fourth Generation Languages

- First generation languages
  - Machine languages
- Second generation languages
  - Assemblers
- Third generation languages
  - High-level languages (COBOL, FORTRAN, C++, Java)

# Fourth Generation Languages (contd)

- Fourth generation languages (4GLs)
  - One 3GL statement is equivalent to 5–10 assembler statements
  - Each 4GL statement was intended to be equivalent to 30 or even 50 assembler statements

# Fourth Generation Languages (contd)

- It was hoped that 4GLs would
  - Speed up application-building
  - Result in applications that are easy to build and quick to change
    - Reducing maintenance costs
  - Simplify debugging
  - Make languages user friendly
    - Leading to end-user programming

- Achievable if 4GL is a user friendly, very high-level language

# Fourth Generation Languages (contd)

- Example
  - See Just in Case You Wanted to Know  Box 15.2

- The power of a nonprocedural language, and the price

# Productivity Increases with a 4GL?

- The picture is not uniformly rosy
- Playtex used ADF, obtained an 80 to 1 productivity increase over COBOL
  - However, Playtex then used COBOL for later applications
- 4GL productivity increases of 10 to 1 over COBOL have been reported
  - However, there are plenty of reports of bad experiences

# Actual Experiences with 4GLs

- Many 4GLs are supported by powerful CASE environments
  - This is a problem for organizations at CMM level 1 or 2
  - Some reported 4GL failures are due to the underlying CASE environment

# Actual Experiences with 4GLs (contd)

- Attitudes of 43 organizations to 4GLs
  - Use of 4GL reduced users' frustrations
  - Quicker response from DP department
  - 4GLs are slow and inefficient, on average
  - Overall, 28 organizations using 4GL for over 3 years felt that the benefits outweighed the costs

# Fourth Generation Languages (contd)

- Market share
  - No one 4GL dominates the software market
  - There are literally hundreds of 4GLs
  - Dozens with sizable user groups
  - Oracle, DB2, and PowerBuilder are extremely popular
- Reason
  - No one 4GL has all the necessary features
- Conclusion
  - Care has to be taken in selecting the appropriate 4GL

# Dangers of a 4GL

- End-user programming
  - Programmers are taught to mistrust computer output
  - End users are taught to believe computer output
  - An end-user updating a database can be particularly dangerous

# Dangers of a 4GL (contd)

- Potential pitfalls for management
  - Premature introduction of a CASE environment
  - Providing insufficient training for the development team
  - Choosing the wrong 4GL

# 15.3 Good Programming Practice

- Use of *consistent* and *meaningful* variable names
  - "Meaningful" to future maintenance programmers
  - "Consistent" to aid future maintenance programmers

# 15.3.1 Use of Consistent and Meaningful Variable Names

- A code artifact includes the variable names
  `freqAverage, frequencyMaximum, minFr, frqncyTotl`

- A maintenance programmer has to know if `freq, frequency, fr, frqncy` all refer to the same thing
  - If so, use the identical word, preferably `frequency`, perhaps `freq` or `frqncy`, but *not* `fr`
  - If not, use a different word (e.g., `rate`) for a different quantity

# Consistent and Meaningful Variable Names

- We can use `frequencyAverage, frequencyMaximum, frequencyMinimum, frequencyTotal`
- We can also use `averageFrequency, maximumFrequency, minimumFrequency, totalFrequency`
- But all four names must come from the same set

# 13.3.2 The Issue of Self-Documenting Code

- Self-documenting code is exceedingly rare

- The key issue: Can the code artifact be understood easily and unambiguously by
  - The SQA team
  - Maintenance programmers
  - All others who have to read the code

# Self-Documenting Code Example

- Example:
  - Code artifact  contains the variable `xCoordinateOfPositionOfRobotArm`
  - This is abbreviated to `xCoord`
  - This is fine, because the entire module deals with the movement of the robot arm
  - But does the maintenance programmer know this?

# Prologue Comments

- Minimal prologue comments for a code artifact

The name of the code artifact
A brief description of what the code artifact does
The programmer's name
The date the code artifact was coded
The date the code artifact was approved
The name of the person who approved the code artifact
The arguments of the code artifact
A list of the name of each variable of the code artifact, preferably in alphabetical order, and a brief description of its use
The names of any files accessed by this code artifact
The names of any files changed by this code artifact
Input–output, if any
Error-handling capabilities
The name of the file containing test data (to be used later for regression testing)
A list of each modification made to the code artifact, the date the modification was made, and who approved the modification
Any known faults

Figure 15.1

# Other Comments

- Suggestion
  - Comments are essential whenever the code is written in a non-obvious way, or makes use of some subtle aspect of the language
- Nonsense!
  - Recode in a clearer way
  - We must never promote/excuse poor programming
  - However, comments can assist future maintenance programmers

# 15.3.3 Use of Parameters

- There are almost no genuine constants
- One solution:
  - Use `const` statements (C++), or
  - Use `public static final` statements (Java)
- A better solution:
  - Read the values of “constants” from a parameter file

# 13.3.4 Code Layout for Increased Readability

- Use indentation
- Better, use a pretty-printer
- Use plenty of blank lines
  - To break up big blocks of code

# 15.3.5 Nested `if` Statements

- Example
  - A map consists of two squares. Write code to determine whether a point on the Earth's surface lies in `mapSquare1` or `mapSquare2`, or is not on the map

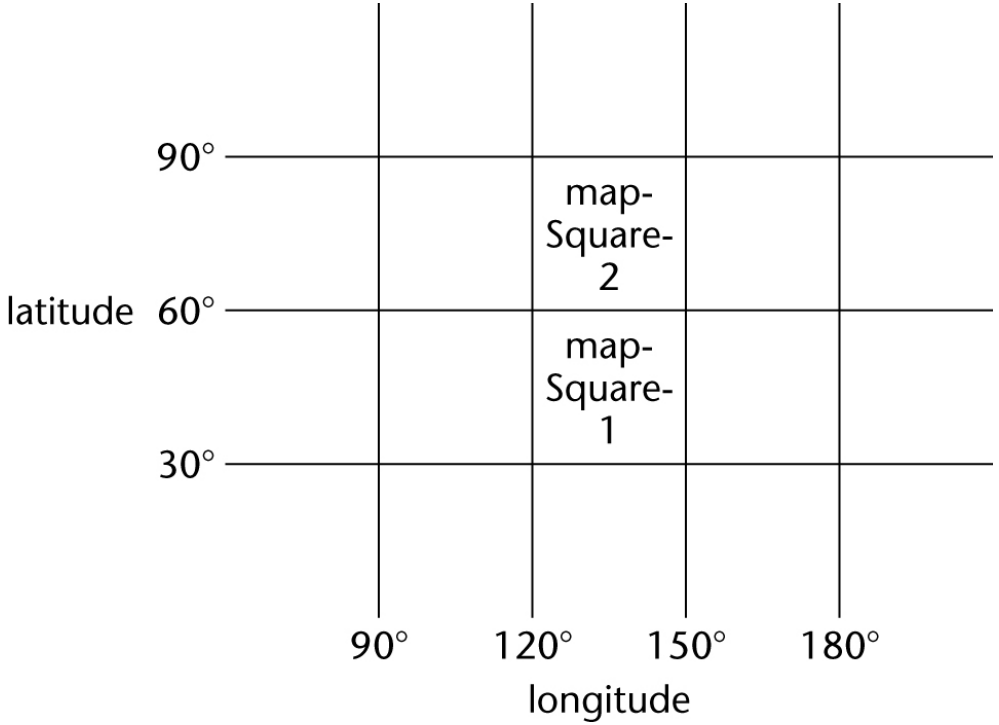


Figure 15.2

# Nested `if` Statements (contd)

- Solution 1. Badly formatted

```
if (latitude > 30 && longitude > 120) {if (latitude <= 60 && longitude <= 150)
mapSquareNo = 1; else if (latitude <= 90 && longitude <= 150) mapSquareNo = 2
else print "Not on the map";} else print "Not on the map";
```

Figure 15.3

# Nested `if` Statements (contd)

- Solution 2. Well-formatted, badly constructed

```
if (latitude > 30 && longitude > 120)
{
   if (latitude <= 60 && longitude <= 150)
      mapSquareNo = 1;
   else
      if (latitude <= 90 && longitude <= 150)
         mapSquareNo = 2;
      else
         print "Not on the map";
}
else
   print "Not on the map";
```

Figure 15.4

# Nested `if` Statements (contd)

- Solution 3. Acceptably nested

```
if (longitude > 120 && longitude <= 150 && latitude > 30 && latitude <= 60)
   mapSquareNo = 1;
else
   if (longitude > 120 && longitude <= 150 && latitude > 60 && latitude <= 90)
      mapSquareNo = 2;
   else
      print "Not on the map";
```

Figure 15.5

# Nested `if` Statements (contd)

- A combination of `if-if` and `if-else-if` statements is usually difficult to read

- Simplify: The **if-if** combination

```
if <condition1>
        if <condition2>
```

is frequently equivalent to the single condition

```
if <condition1> && <condition2>
```

# Nested `if` Statements (contd)

- Rule of thumb
  - `if` statements nested to a depth of greater than three should be avoided as poor programming practice

# 15.4 Programming Standards

- Standards can be both a blessing and a curse

- Modules of coincidental cohesion arise from rules like
  - "Every module will consist of between 35 and 50 executable statements"

- Better
  - "Programmers should consult their managers before constructing a module with fewer than 35 or more than 50 executable statements"

# Remarks on Programming Standards

- No standard can ever be universally applicable
- Standards imposed from above will be ignored
- Standard must be checkable by machine

# Examples of Good Programming Standards

- "Nesting of `if` statements should not exceed a depth of 3, except with prior approval from the team leader"

- "Modules should consist of between 35 and 50 statements, except with prior approval from the team leader"

- "Use of `goto`s should be avoided. However, with prior approval from the team leader, a forward `goto` may be used for error handling"

# Remarks on Programming Standards (contd)

- The aim of standards is to make maintenance easier
  - If they make development difficult, then they must be modified
  - Overly restrictive standards are counterproductive
  - The quality of software suffers

# 15.5 Code Reuse

- Code reuse is the most common form of reuse
- However, artifacts from all workflows can be reused
  - For this reason, the material on reuse appears in Chapter 8, and not here

# 15.6 Integration

- The approach up to now:
  - Implementation followed by integration
- This is a poor approach
- Better:
  - Combine implementation and integration methodically

# Product with 13 Modules

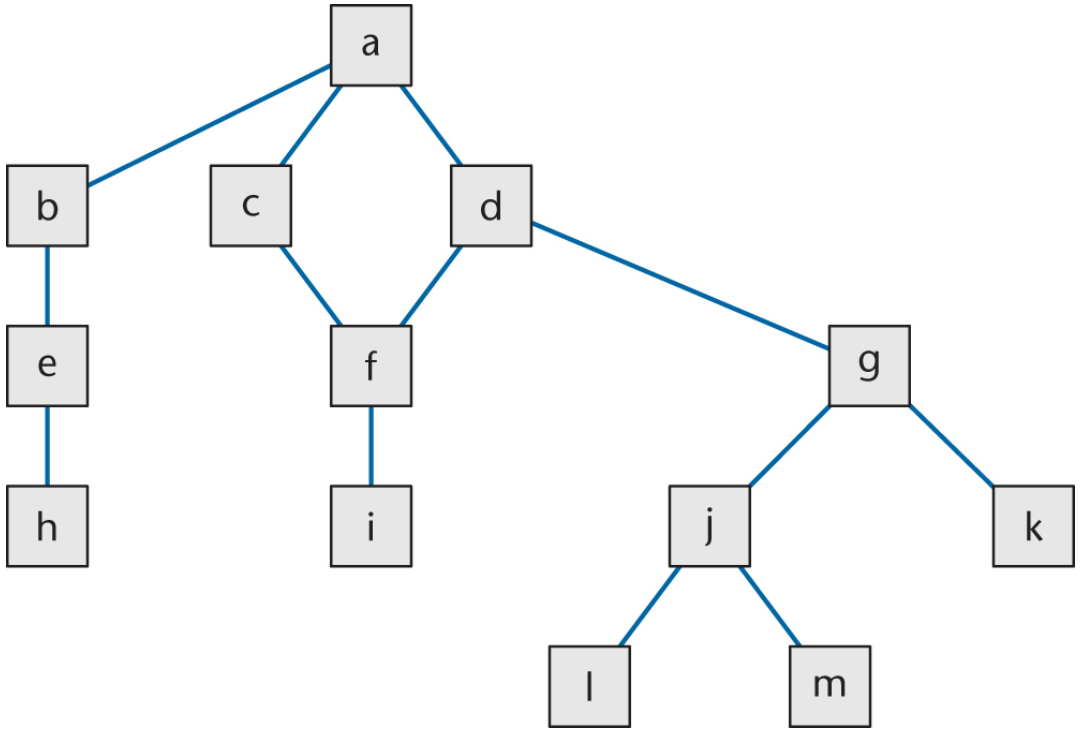


Figure 15.6

# Implementation, Then Integration

- Code and test each code artifact separately
- Link all 13 artifacts together, test the product as a whole

# Drivers and Stubs

- To test artifact `a`, artifacts `b, c, d` must be stubs
  - An empty artifact, or
  - Prints a message ("`Procedure radarCalc called`"), or
  - Returns precooked values from preplanned test cases

- To test artifact `h` on its own requires a driver, which calls it
  - Once, or
  - Several times, or
  - Many times, each time checking the value returned

- Testing artifact `d` requires a driver and two stubs

# Implementation, Then Integration (contd)

- Problem 1
  - Stubs and drivers must be written, then thrown away after unit testing is complete
- Problem 2
  - Lack of fault isolation
  - A fault could lie in *any* of the 13 artifacts or 13 interfaces
  - In a large product with, say, 103 artifacts and 108 interfaces, there are 211 places where a fault might lie

# Implementation, Then Integration (contd)

- Solution to both problems
  - Combine unit and integration testing

# 15.6.1 Top-down Integration

- If code artifact `mAbove` sends a message to artifact `mBelow`, then `mAbove` is implemented and integrated before `mBelow`

- One possible top-down ordering is
  - `a, b, c, d, e, f, g, h, i, j, k, l ,m`

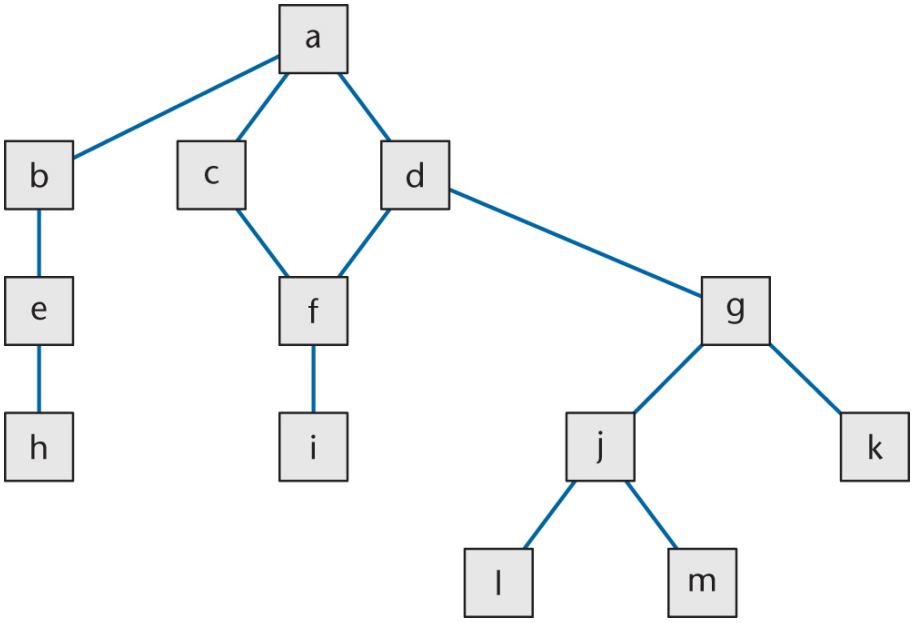


Figure 15.6 (again)

# Top-down Integration (contd)

- Another possible top-down ordering is

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

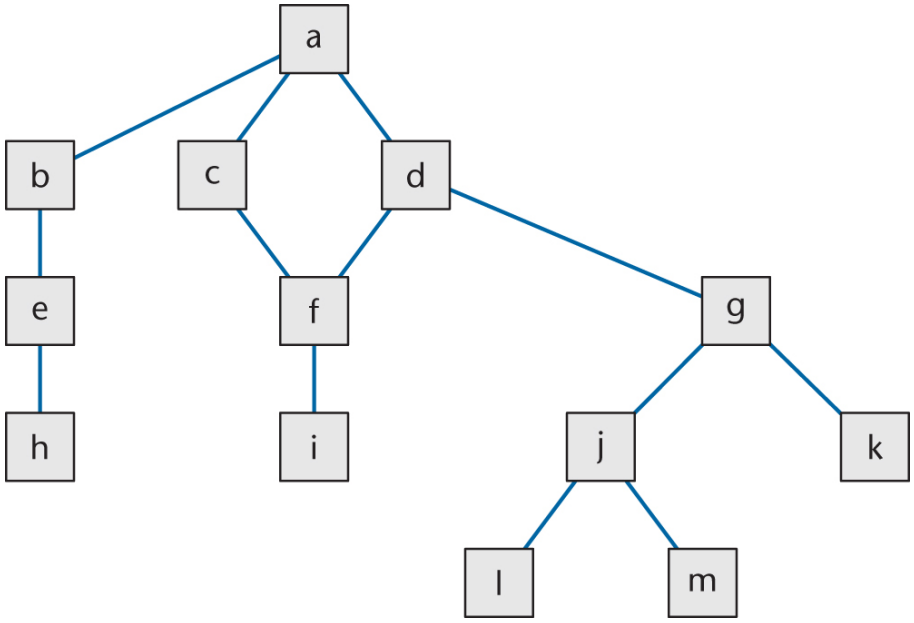


Figure 15.6 (again)

# Top-down Integration (contd)

- Advantage 1: Fault isolation
  - A previously successful test case fails when `mNew` is added to what has been tested so far
    - The fault must lie in `mNew` or the interface(s) between `mNew` and the rest of the product
- Advantage 2: Stubs are not wasted
  - Each stub is expanded into the corresponding complete artifact at the appropriate step

# Top-down Integration (contd)

- Advantage 3: Major design flaws show up early
- Logic artifacts include the decision-making flow of control
  - In the example, artifacts `a, b, c, d, g, j`
- Operational artifacts 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 (contd)

- Problem 1
  - Reusable artifacts are not properly tested
  - Lower level (operational) artifacts are not tested frequently
  - The situation is aggravated if the product is well designed

- Defensive programming (fault shielding)
  - Example:

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

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

# 15.6.2 Bottom-up Integration

- If code artifact `mAbove` calls code artifact `mBelow`, then `mBelow` is implemented and integrated before `mAbove`

- One possible bottom-up ordering is

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

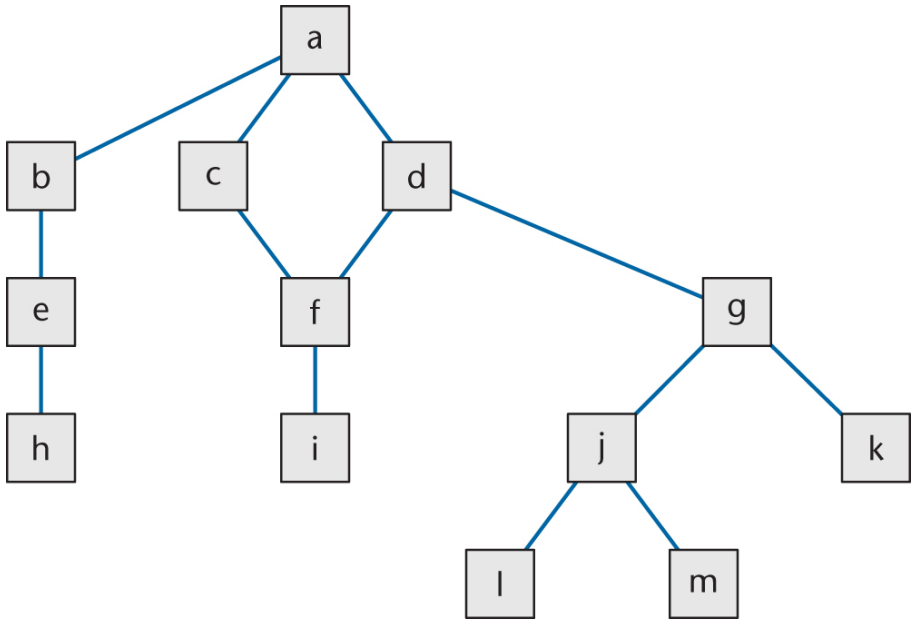


Figure 15.6 (again)

# 15.6.2 Bottom-up Integration

- Another possible bottom-up ordering is

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

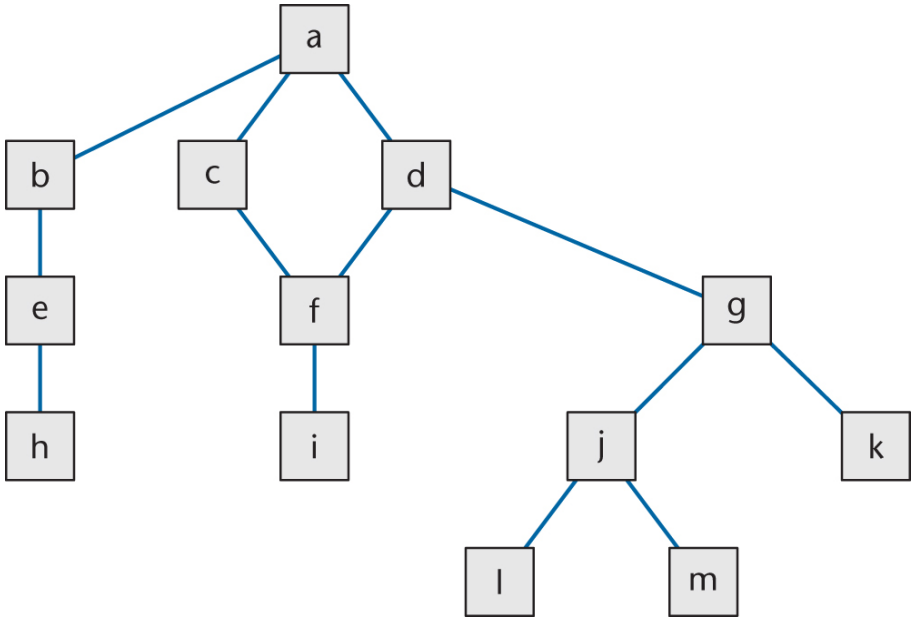


Figure 15.6 (again)

# Bottom-up Integration (contd)

- Advantage 1
  - Operational artifacts are thoroughly tested
- Advantage 2
  - Operational artifacts are tested with drivers, not by fault shielding, defensively programmed artifacts
- Advantage 3
  - Fault isolation

# Bottom-up Integration (contd)

- Difficulty 1
  - Major design faults are detected late
- Solution
  - Combine top-down and bottom-up strategies making use of their strengths and minimizing their weaknesses

# 15.6.3 Sandwich Integration

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

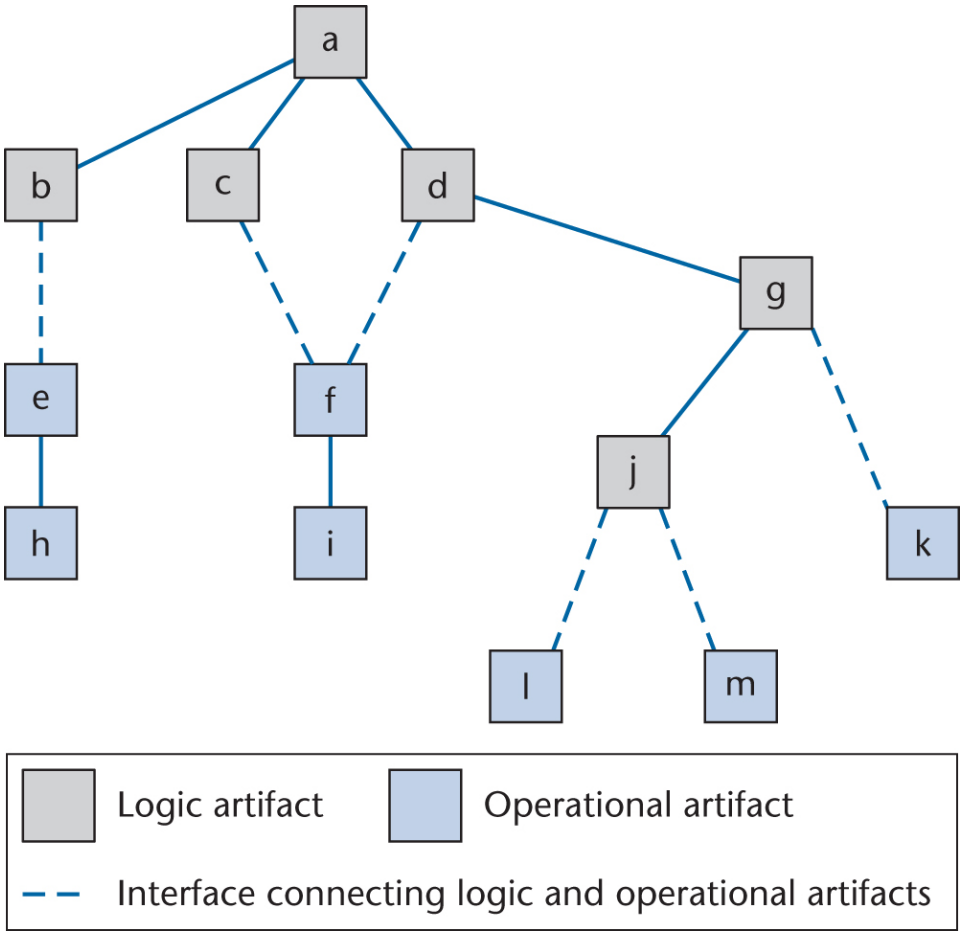


Figure 15.7

# Sandwich Integration (contd)

- Advantage 1
  - Major design faults are caught early
- Advantage 2
  - Operational artifacts are thoroughly tested
  - They may be reused with confidence
- Advantage 3
  - There is fault isolation at all times

# Summary

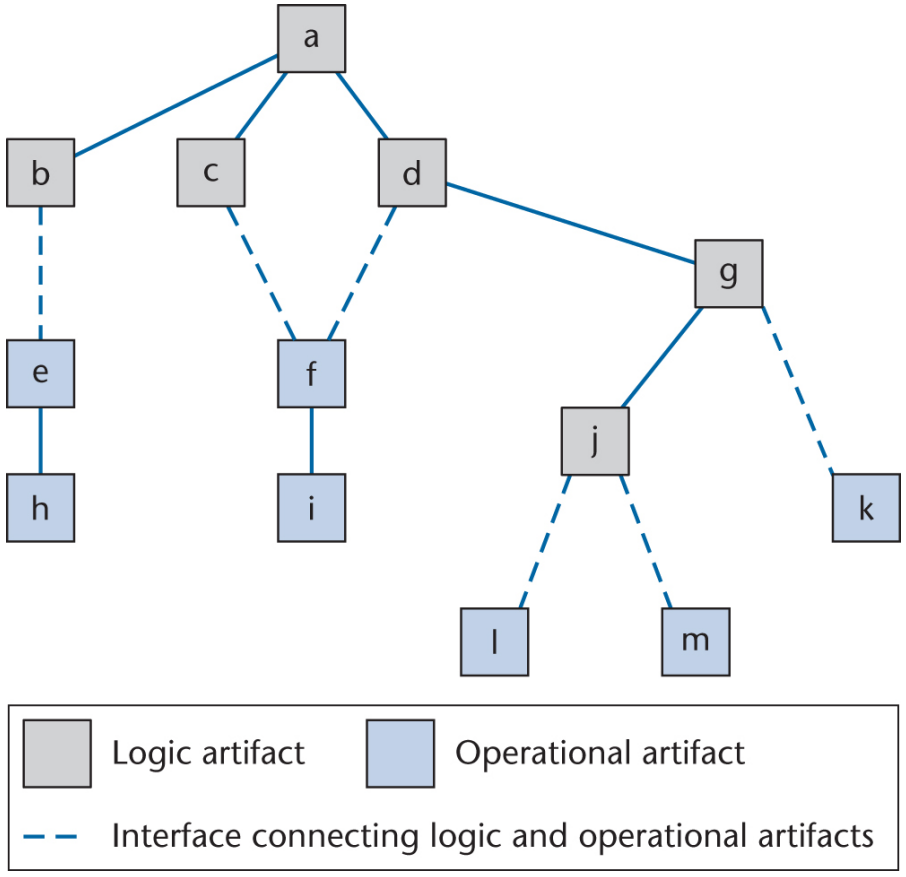


Figure 15.8

# Feasibility of Testing to Code (contd)

- A path can be tested only if it is present

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

(a)

- A programmer who omits the test for d = 0 in the code probably is unaware of the possible danger

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

(b)

Figure 15.12

# Feasibility of Testing to Code (contd)

- Criterion “exercise all paths” is not *reliable*
  - Products exist for which some data exercising a given path detect a fault, and other data exercising the same path do not

# Problem with Complexity Metrics

- Complexity metrics, as especially cyclomatic complexity, have been strongly challenged on
  - Theoretical grounds
  - Experimental grounds, and
  - Their high correlation with LOC

- Essentially we are measuring lines of code, not complexity

# Code Walkthroughs and Inspections

- Code reviews lead to rapid and thorough fault detection
  - Up to 95% reduction in maintenance costs

# 15.25 Metrics for the Implementation Workflow

- The five basic metrics, plus
  - Complexity metrics
- Fault statistics are important
  - Number of test cases
  - Percentage of test cases that resulted in failure
  - Total number of faults, by types
- The fault data are incorporated into checklists for code inspections

# 15.26 Challenges of the Implementation Workflow

- Management issues are paramount here
  - Appropriate CASE tools
  - Test case planning
  - Communicating changes to all personnel
  - Deciding when to stop testing

# Challenges of the Implementation Workflow (contd)

- Code reuse needs to be built into the product from the very beginning
  - Reuse must be a client requirement
  - The software project management plan must incorporate reuse
- Implementation is technically straightforward
  - The challenges are managerial in nature

# Challenges of the Implementation Phase (contd)

- Make-or-break issues include:
  - Use of appropriate CASE tools
  - Test planning as soon as the client has signed off the specifications
  - Ensuring that changes are communicated to all relevant personnel
  - Deciding when to stop testing