OBJECT-ORIENTED AND CLASSICAL SOFTWARE ENGINEERING

EIGHTH EDITION, WCB/MCGRAW-HILL, 2011

STEPHEN R. SCHACH

Edited by Hyungbae Park

CHAPTER 15

IMPLEMENTATION

Overview

- ☐ Choice of programming language
- ☐ Fourth generation languages
- ☐ Good programming practice
- ☐ Coding standards
- ☐ Code reuse
- ☐ Integration
- ☐ The implementation workflow
- ☐ The implementation workflow: The MSG Foundation case study

Implementation

- Real-life products are generally too large to be implemented by a single programmer

- This chapter therefore deals with programming-in-the-many

15.1 Choice of Programming Language (contd)

- The language is usually specified in the contract

- But what if the contract specifies that
    - The product is to be implemented in the "most suitable" programming language

- What language should be chosen?

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

Productivity Increases with a 4GL?

- 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

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

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` acceptable, but *not* `fr`
  - If not, use a different word (e.g., `rate`) for a different quantity
  - Do not use two different name to denote the identical concept, both `average` and `mean` should not be used in the same program

Consistent and Meaningful Variable Names

□ Consistency

- ◻ **If one variable name is** `frequencyMinimum`**, then the name** `minimumFrequency` **would be confusing**

- ◻ **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**

15.3.2 The Issue of Self-Documenting Code

- "I write self-documenting code."
  - Implication is that their variable names are chosen so carefully and their code crafted exquisitely that there is no need for comments

- 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`
    - Few programmers are prepared to use a 31-character variable name, especially if that name is used frequently
  - 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

Comments examples

```
HelloWorld.java
1  ////////////////////////////////////////////////////////////////////////////////
2  //                      ALL STUDENTS COMPLETE THESE SECTIONS
3  // Title:               HelloWorld
4  // Files:               HelloWorld.java
5  // Semester:            Fall 2014
6  //
7  // Author:              Deb Deppeler deppeler@cs.wisc.edu
8  // CS Login:            deppeler
9  // Lecturer's Name:     DEPPELER
10 // Lab Section:         310
11 //
12 //                      PAIR PROGRAMMERS COMPLETE THIS SECTION
13 // Pair Partner:        Hobbes LeGault legault@cs.wisc.edu
14 // CS Login:            legault
15 // Lecturer's Name:     LEGAULT
16 // Lab Section:         330
17 //
18 //                      STUDENTS WHO GET HELP FROM ANYONE OTHER THAN THEIR PARTNER
19 // Credits:             Lab #1 Eclipse tutorial instructions
20 ///////////////////////////// 80 columns wide //////////////////////////////////
21
22 /** Application that displays "Hello, World!" to console window. */
23 public class HelloWorld {
24
25     /**
26      * Program execution begins here.
27      * @param args UNUSED
28      */
29     public static void main(String[] args) {
30         System.out.println("Hello, World!");
31     }
32 }
33
```

file header comment

class header comment

method header comment

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

15.3.4 Code Layout for Increased Readability

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

```cpp
#include<iostream>
using namespace std;
int main()
{
    int range, first = 0, second = 1, fibonicci;

    cout << "Enter the Range for Terms of Fibonacci Sequence " << endl;
    cin >> range;

    cout << "Fibonicci Series upto " << range << " Terms "<<endl<< endl;

    for ( int c = 0 ; c < range ; c++ )
    {
        if ( c <= 1 )
            fibonicci = c;
        else
        {
            fibonicci = first + second;
            first = second;
            second = fibonicci;
        }
        cout << fibonicci <<" ";
    }
return 0;
}// end of main body
```

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
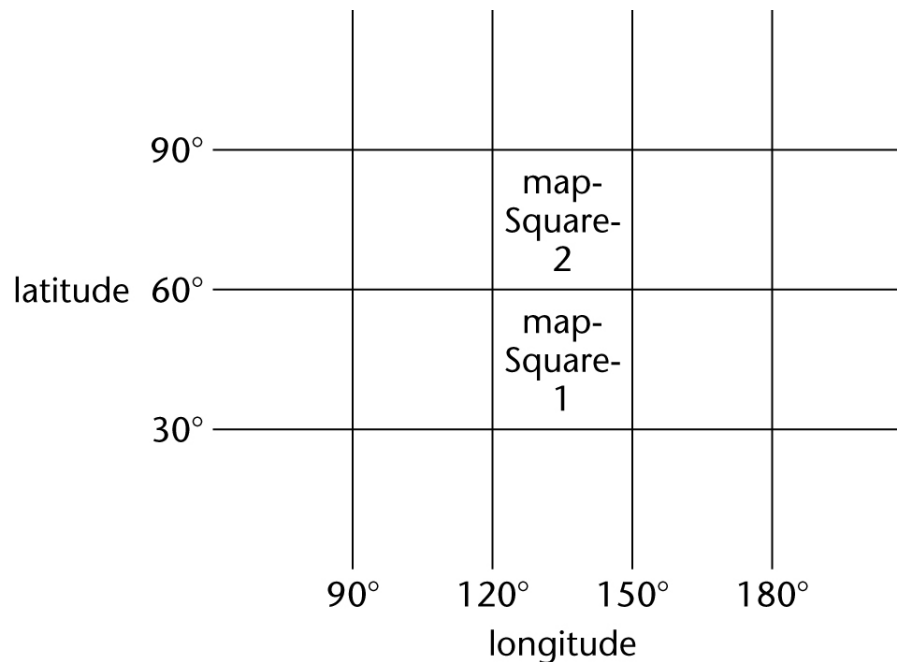

Figure 15.2

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"

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

Remarks on Programming Standards

☐ No standard can ever be universally applicable

☐ Standards imposed from above will be ignored

☐ Standard must be checkable by machine

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:
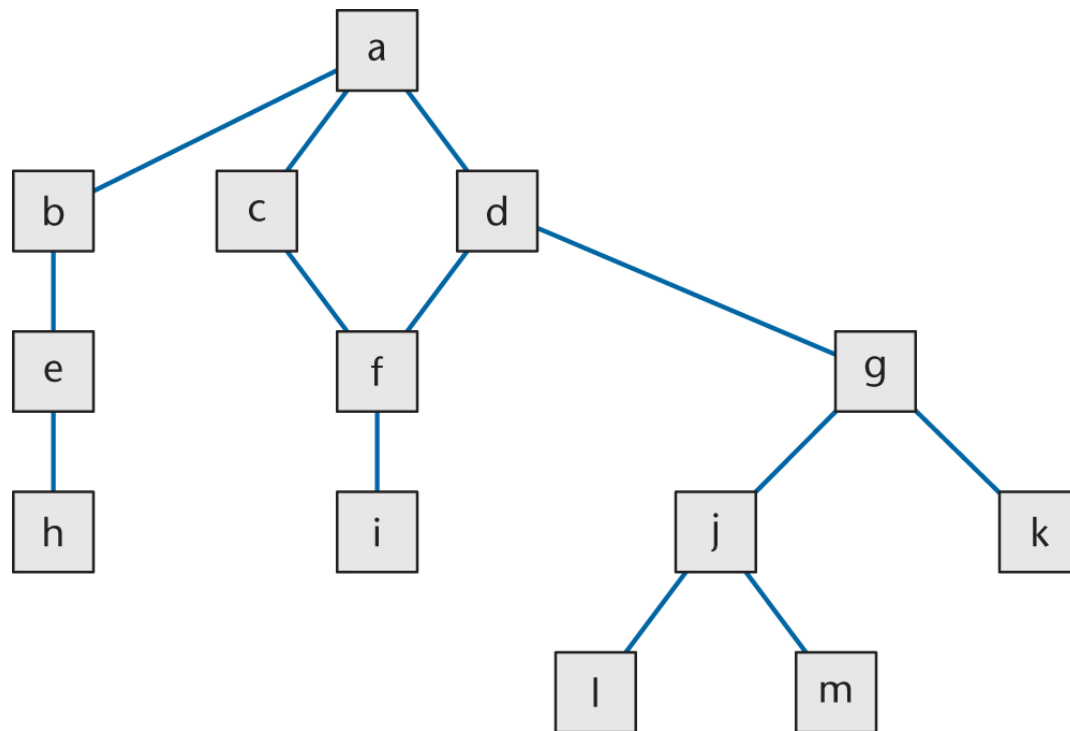 - Combine implementation and integration methodically

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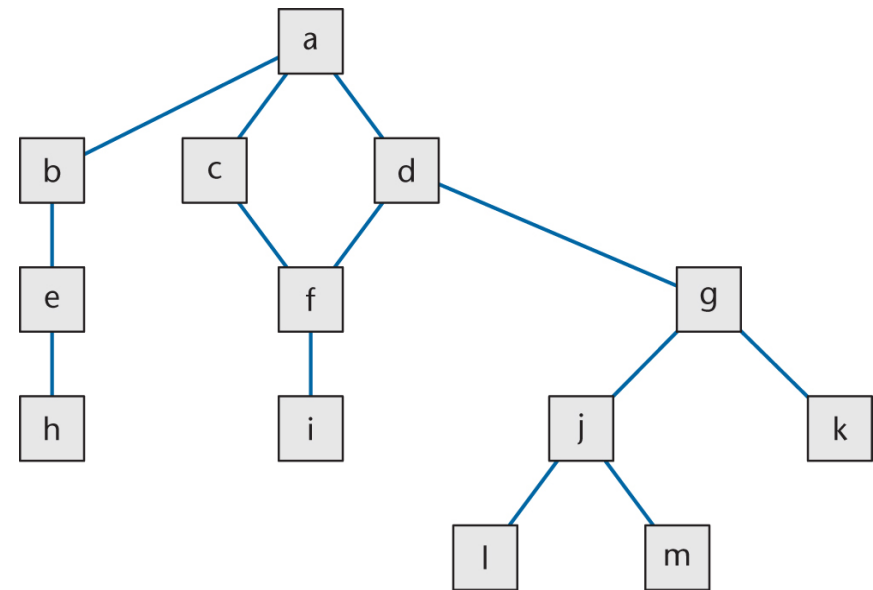
▪ `a, b, c, d, e, f, g, h, i, j, k, l ,m`

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
```
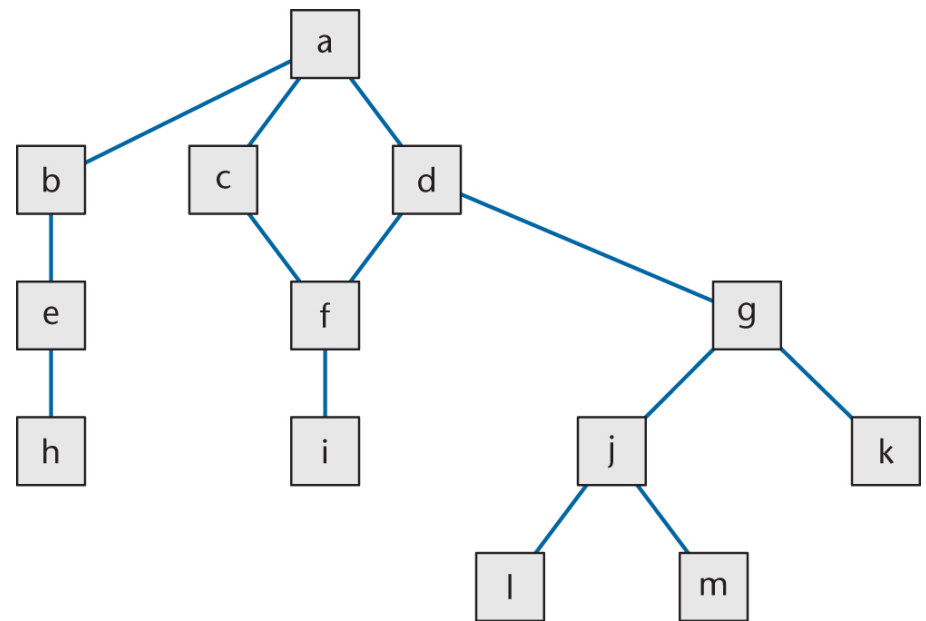

Figure 15.6 (again)

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
  ```
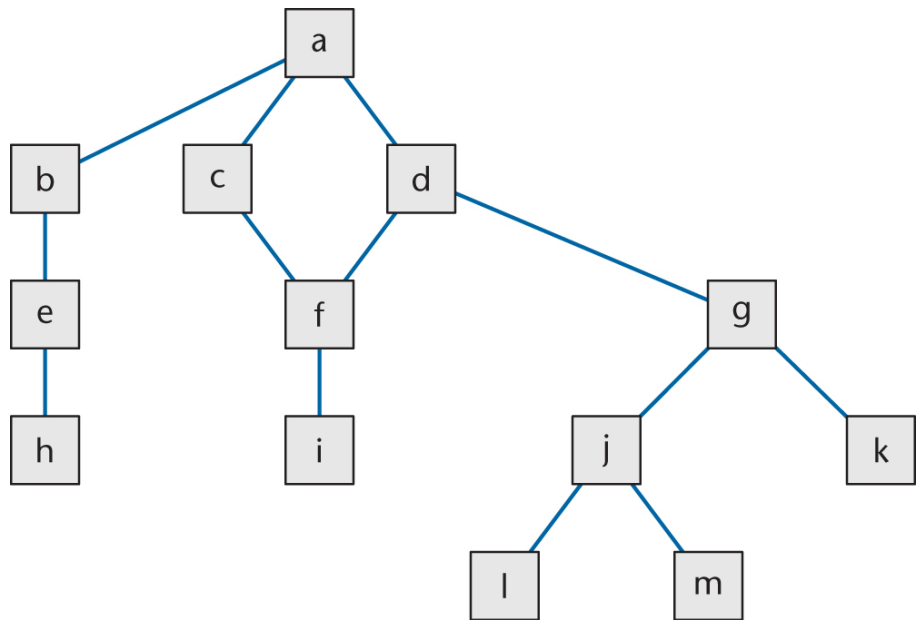
Figure 15.6 (again)

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]
```
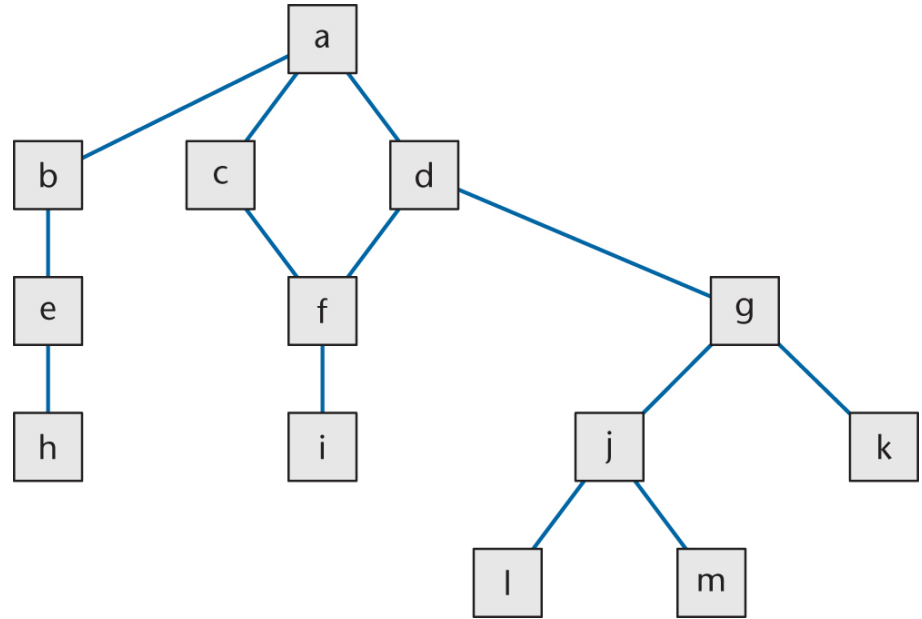

Figure 15.6 (again)

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: incorporate the decision-making flow of control aspects of the product

- Operational artifacts: perform the actual operations of the product

- Logic artifacts are integrated top-down

- Operational artifacts are integrated bottom-up

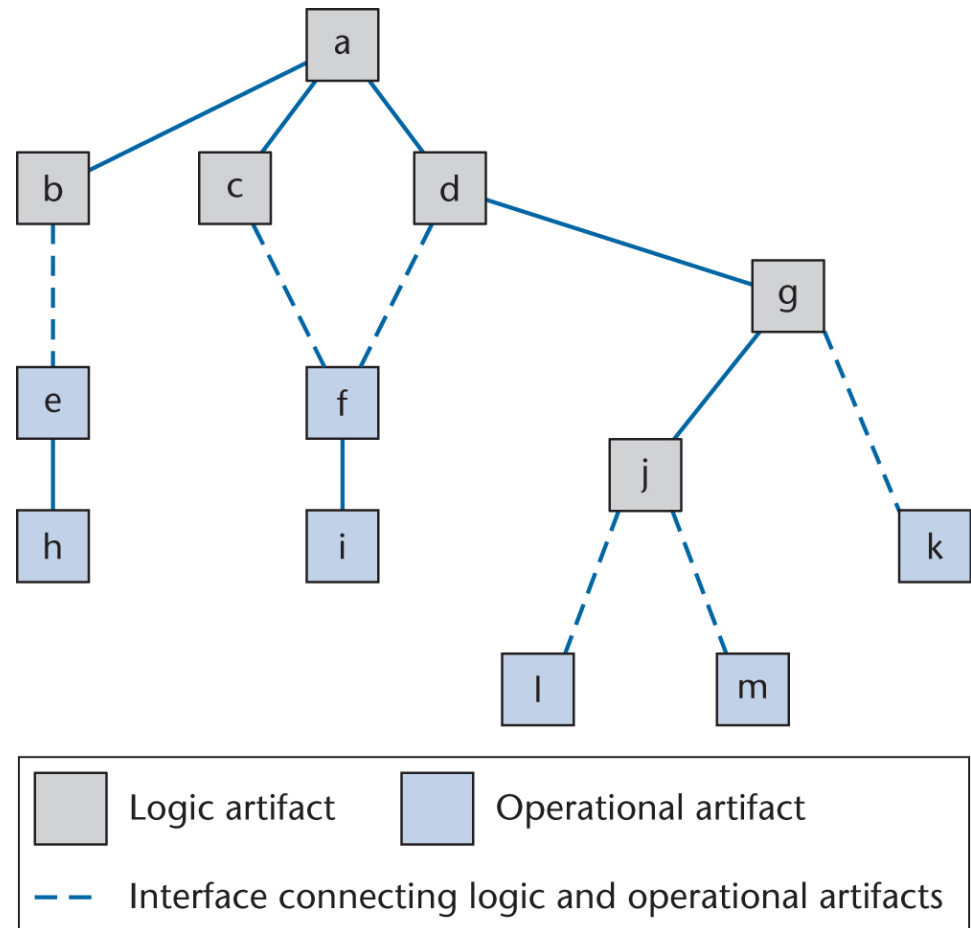- Finally, the interfaces between the two groups are tested

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

15.6.4 Integration of Object-Oriented Products

- Object-oriented implementation and integration
    - Almost always sandwich implementation and integration
    - Objects are integrated bottom-up
    - Other artifacts are integrated top-down

15.7 The Implementation Workflow

- The aim of the implementation workflow is to implement the target software product

- A large product is partitioned into *subsystems*
  - Implemented in parallel by coding teams

- Subsystems consist of *components* or *code artifacts*

The Implementation Workflow (contd)

- Once the programmer has implemented an artifact, he or she *unit tests* it

- Then the module is passed on to the SQA group for further testing
  - This testing is part of the test workflow