Object-Oriented and Classical Software Engineering

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
- The test workflow: Implementation

Overview (contd)

- Test case selection
- Black-box unit-testing techniques
- Black-box test cases: The MSG Foundation case study
- Glass-box unit-testing technique
- Code walkthroughs and inspections
- Comparison of unit-testing techniques
- Cleanroom
- Potential problems when testing objects
- Management aspects of unit testing

Overview (contd)

- When to rewrite rather than debug a module
- Integration testing
- Product testing
- Acceptance testing
- The test workflow: The MSG Foundation case study
- CASE tools for implementation
- Metrics for the implementation workflow
- Challenges of the implementation workflow

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

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

15.3.2 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

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

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

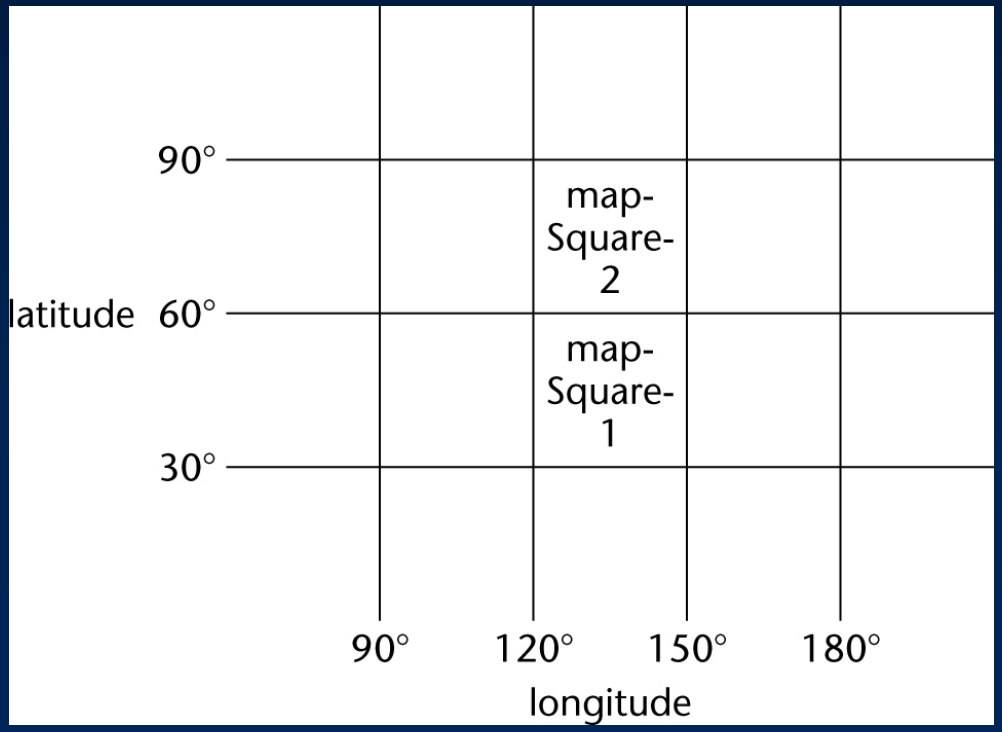


Figure 15.2

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

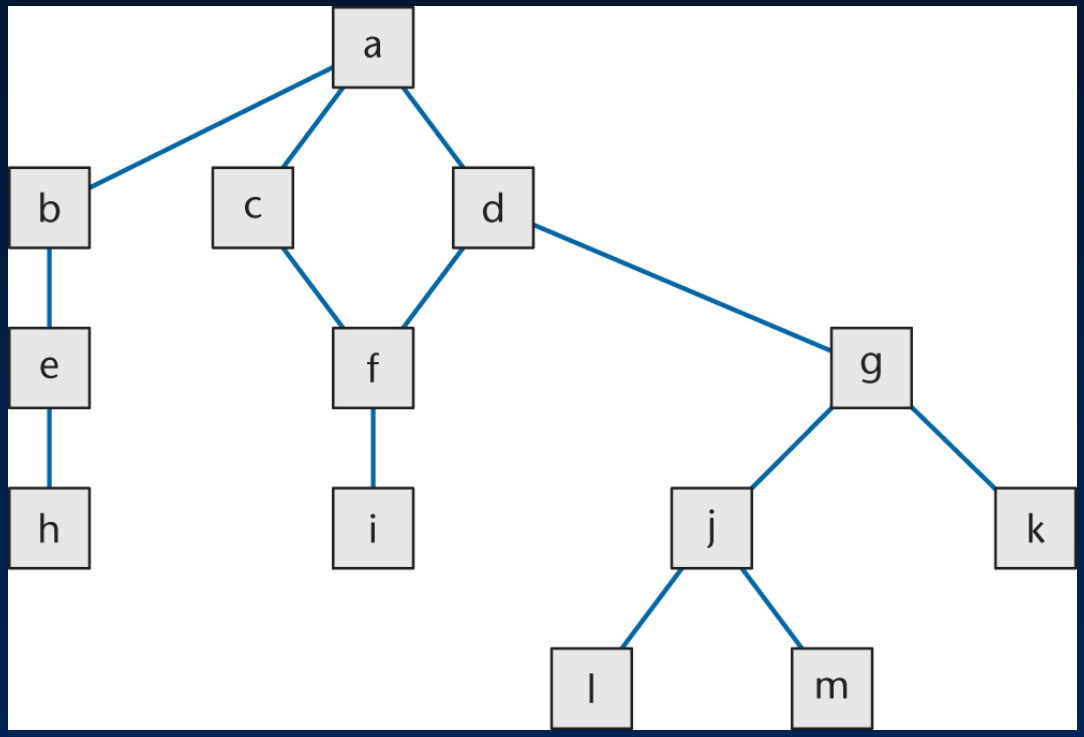


Figure 15.6

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`

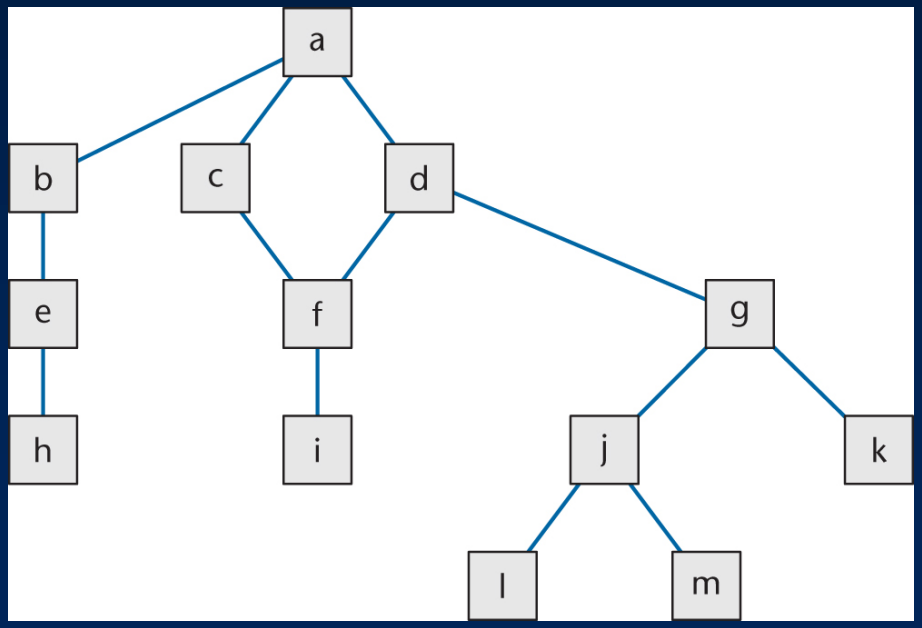


Figure 15.6 (again)

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
```

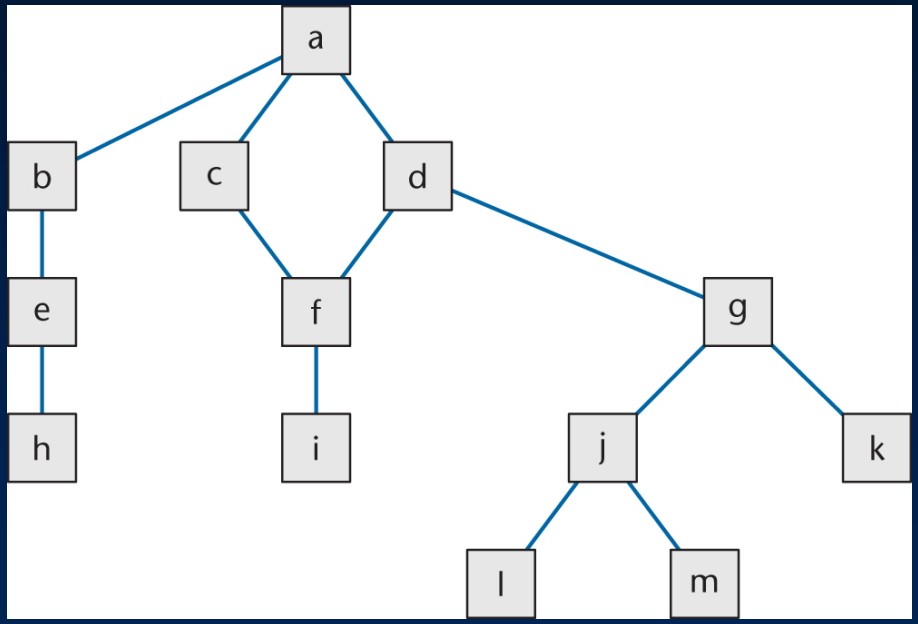


Figure 15.6 (again)

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

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`

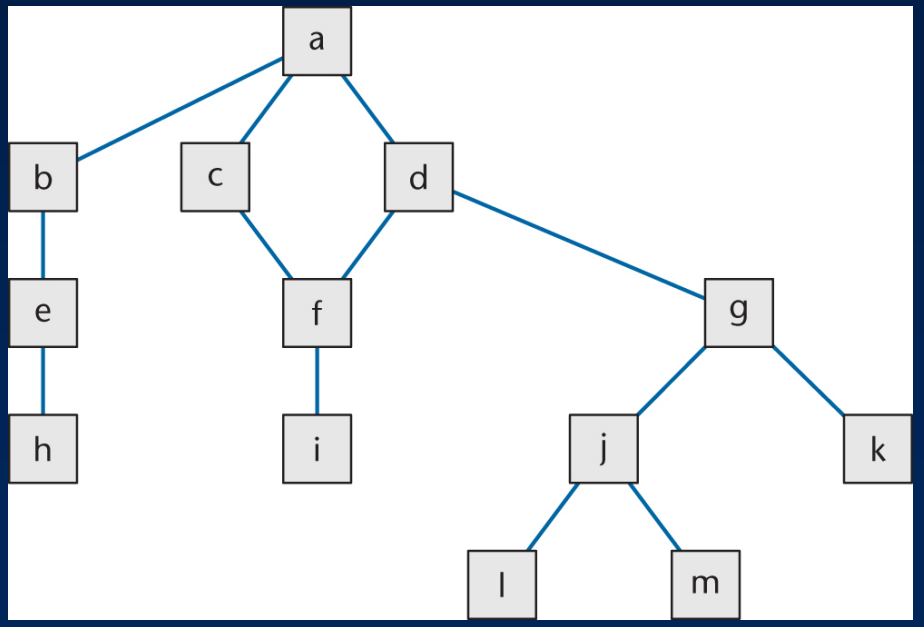


Figure 15.6 (again)

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]
```

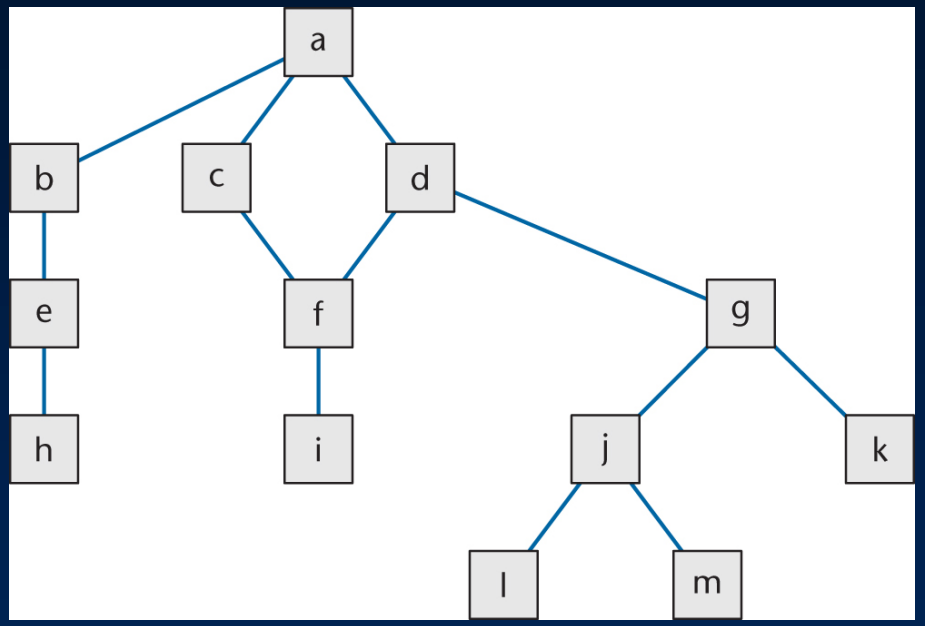


Figure 15.6 (again)

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

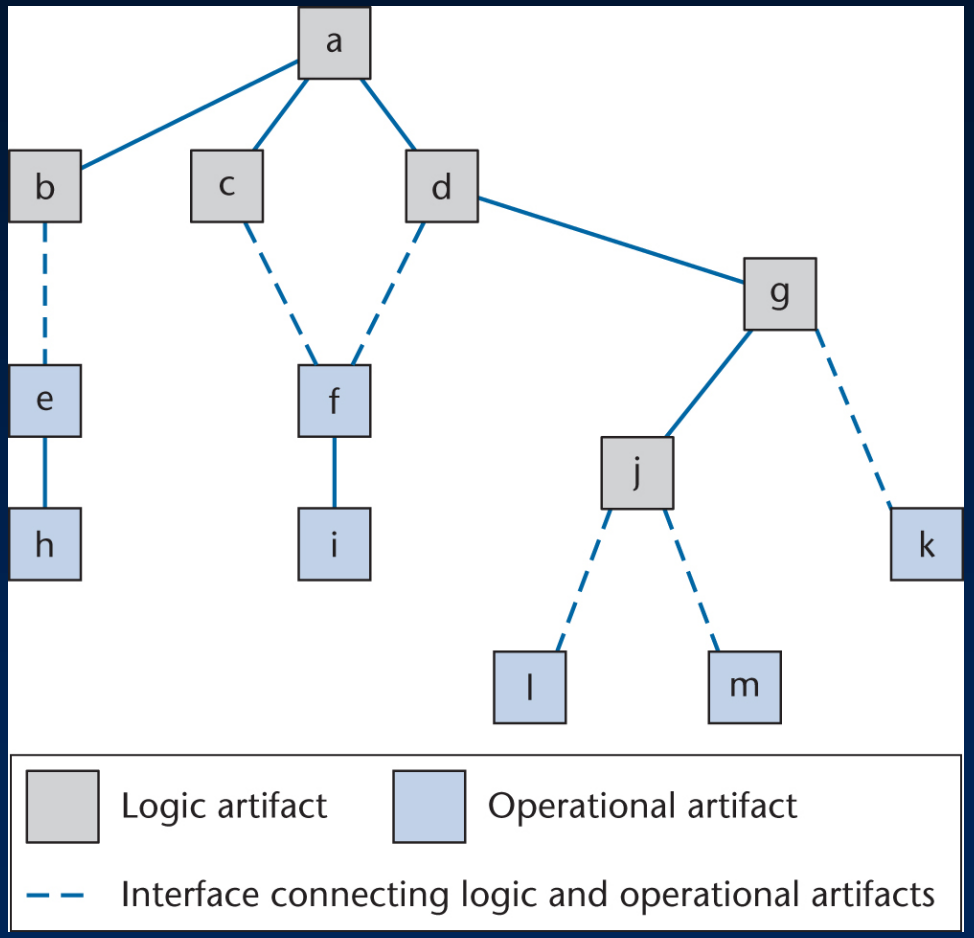


Figure 15.7

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

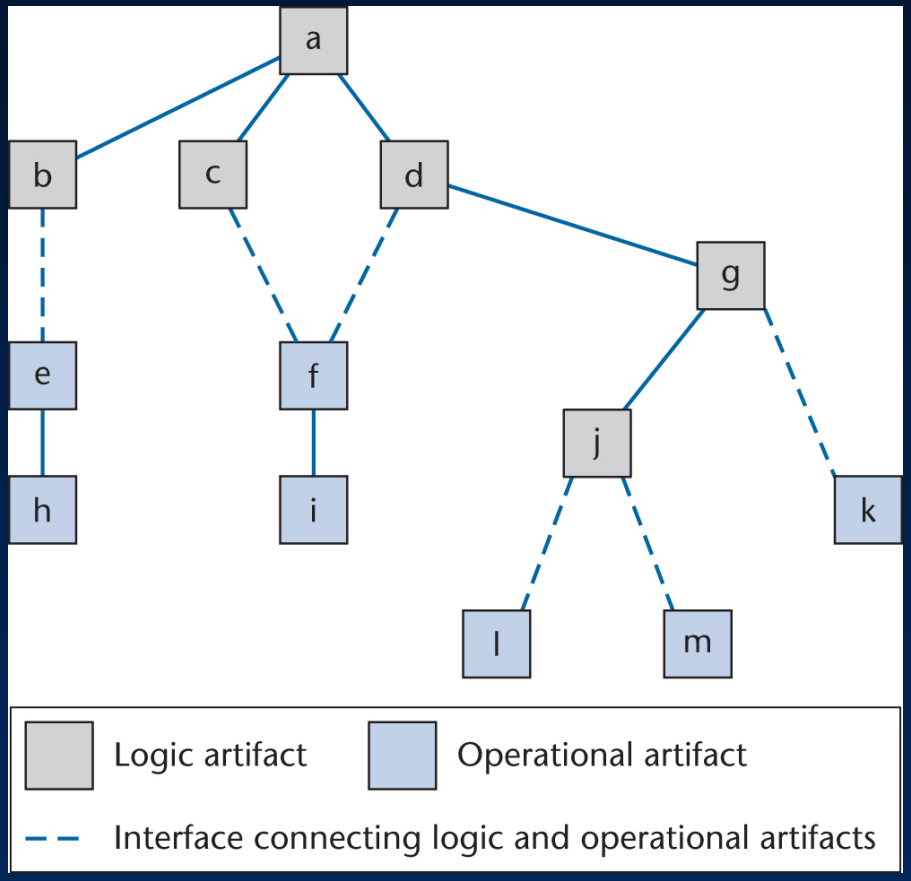


Figure 15.8

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

15.6.5 Management of Integration

- Example:
  - Design document used by programmer P_1 (who coded code object `o1`) shows `o1` sends a message to `o2` passing 4 arguments
  - Design document used by programmer P_2 (who coded code artifact `o2`) states clearly that only 3 arguments are passed to `o2`
- Solution:
  - The integration process must be run by the SQA group
  - They have the most to lose if something goes wrong

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

- Complete implementations in Java and C++ can be downloaded from www.mhhe.com/engcs/schach

15.9 The Test Workflow: Implementation

- Unit testing
  - Informal unit testing by the programmer
  - Methodical unit testing by the SQA group
- There are two types of methodical unit testing
  - Non-execution-based testing
  - Execution-based testing

15.10 Test Case Selection

- Worst way — random testing
  - There is no time to test all but the tiniest fraction of all possible test cases, totaling perhaps 10^{100} or more
- We need a systematic way to construct test cases

15.10.1 Testing to Specifications versus Testing to Code

- There are two extremes to 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

15.10.2 Feasibility of Testing to Specifications

- Example:
  - The specifications for a data processing product include 5 types of commission and 7 types of discount
  - 35 test cases

- We cannot say that commission and discount are computed in two entirely separate artifacts — the structure is irrelevant

Feasibility of Testing to Specifications (contd)

- Suppose the specifications include 20 factors, each taking on 4 values
  - There are 4^{20} or $1.1 \acute{} 10^{12}$ test cases
  - If each takes 30 seconds to run, running all test cases takes more than 1 million years
- The combinatorial explosion makes testing to specifications impossible

15.10.3 Feasibility of Testing to Code

- Each path through a artifact must be executed at least once
  - Combinatorial explosion

Feasibility of Testing to Code (contd)

- Code example:

```
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;
}
```

Figure 15.9

Feasibility of Testing to Code (contd)

- The flowchart has over 10^{12} different paths

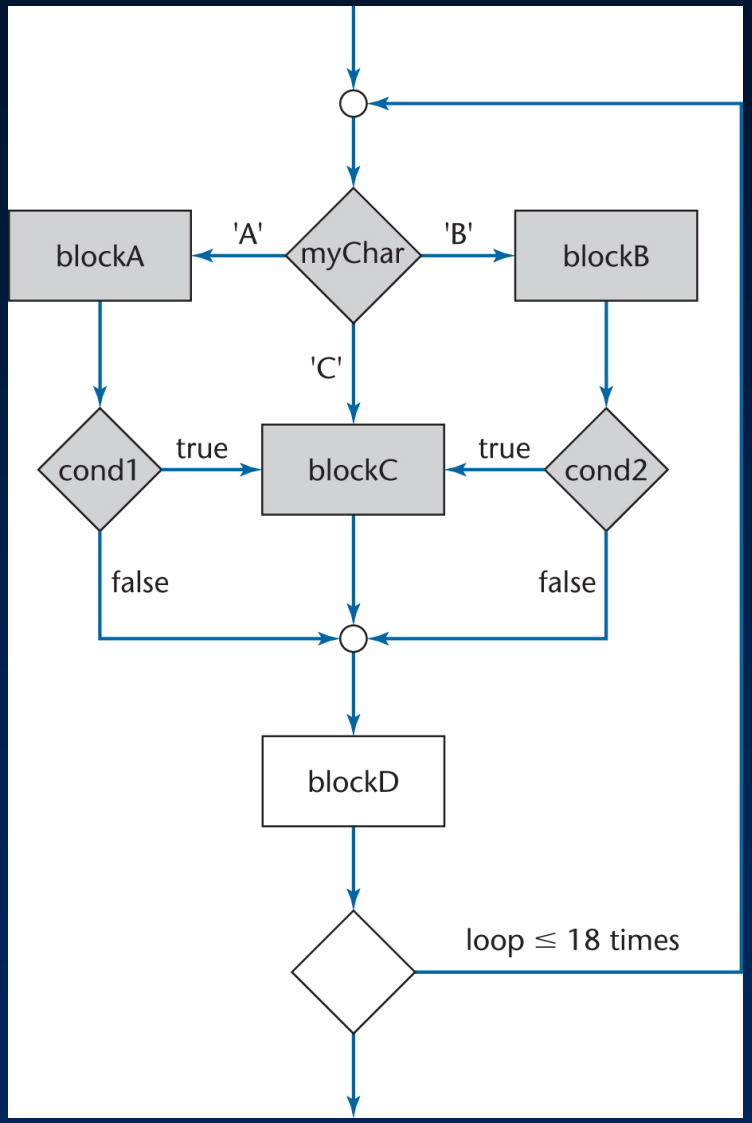


Figure 15.10

Feasibility of Testing to Code (contd)

- 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

Figure 15.11

- We can exercise every path without detecting every fault

Feasibility of Testing to Code (contd)

- 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)

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

15.11 Black-Box Unit-testing Techniques

- Neither exhaustive testing to specifications nor exhaustive testing to code is feasible
- The art of testing:
  - Select a small, manageable set of test cases to
  - Maximize the chances of detecting a fault, while
  - Minimizing the chances of wasting a test case
- Every test case must detect a previously undetected fault

Black-Box Unit-testing Techniques (contd)

- We need a method that will highlight as many faults as possible
  - First black-box test cases (testing to specifications)
  - Then glass-box methods (testing to code)

15.11.1 Equivalence Testing and Boundary Value Analysis

- Example
  - The specifications for a DBMS state that the product must handle any number of records between 1 and 16,383 ($2^{14} - 1$)
  - If the system can handle 34 records and 14,870 records, then it probably will work fine for 8,252 records

- If the system works for any one test case in the range (1..16,383), then it will probably work for any other test case in the range
  - Range (1..16,383) constitutes an *equivalence class*

Equivalence Testing

- Any one member of an equivalence class is as good a test case as any other member of the equivalence class

- Range (1..16,383) defines three different equivalence classes:
  - Equivalence Class 1: Fewer than 1 record
  - Equivalence Class 2: Between 1 and 16,383 records
  - Equivalence Class 3: More than 16,383 records

Boundary Value Analysis

- Select test cases on or just to one side of the boundary of equivalence classes
  - This greatly increases the probability of detecting a fault

Database Example (contd)

• Test case 1: 0 records	Member of equivalence class 1 and adjacent to boundary value
• Test case 2: 1 record	Boundary value
• Test case 3: 2 records	Adjacent to boundary value
• Test case 4: 723 records	Member of equivalence class 2

Database Example (contd)

- Test case 5: 16,382 records — Adjacent to boundary value
- Test case 6: 16,383 records — Boundary value
- Test case 7: 16,384 records — Member of equivalence class 3 and adjacent to boundary value

Equivalence Testing of Output Specifications

- We also need to perform equivalence testing of the output specifications

- Example:

  In 2008, the minimum Social Security (OASDI) deduction from any one paycheck was $0, and the maximum was $6,324

  – Test cases must include input data that should result in deductions of exactly $0 and exactly $6,324

  – Also, test data that might result in deductions of less than $0 or more than $6,324

Overall Strategy

- Equivalence classes together with boundary value analysis to test both input specifications and output specifications
  - This approach generates a small set of test data with the potential of uncovering a large number of faults

15.11.2 Functional Testing

- An alternative form of black-box testing for classical software
  - We base the test data on the functionality of the code artifacts
- Each item of functionality or function is identified
- Test data are devised to test each (lower-level) function separately
- Then, higher-level functions composed of these lower-level functions are tested

Functional Testing (contd)

- In practice, however
  - Higher-level functions are not always neatly constructed out of lower-level functions using the constructs of structured programming
  - Instead, the lower-level functions are often intertwined
- Also, functionality boundaries do not always coincide with code artifact boundaries
  - The distinction between unit testing and integration testing becomes blurred
  - This problem also can arise in the object-oriented paradigm when messages are passed between objects

Functional Testing (contd)

- The resulting random interrelationships between code artifacts can have negative consequences for management
  - Milestones and deadlines can become ill-defined
  - The status of the project then becomes hard to determine

15.12 Black-Box Test Cases: The MSG Foundation Case Study

- Test cases derived from equivalence classes and boundary value analysis

Investment data:

Equivalence classes for itemName.

1. First character not alphabetic	Error
2. < 1 character	Error
3. 1 character	Acceptable
4. Between 1 and 25 characters	Acceptable
5. 25 characters	Acceptable
6. > 25 characters	Error (name too long)

Equivalence classes for itemNumber.

1. Character instead of digit	Error (not a number)
2. < 12 digits	Acceptable
3. 12 digits	Acceptable
4. > 12 digits	Error (too many digits)

Equivalence classes for estimatedAnnualReturn and expectedAnnualOperatingExpenses.

1. < $0.00	Error
2. $0.00	Acceptable
3. $0.01	Acceptable
4. Between $0.01 and $999,999,999.97	Acceptable
5. $999,999,999.98	Acceptable
6. $999,999,999.99	Acceptable
7. $1,000,000,000.00	Error
8. > $1,000,000,000.00	Error
9. Character instead of digit	Error (not a number)

Mortgage information:

Equivalence classes for accountNumber are same as for itemNumber above.

Equivalence classes for last name of mortgagees

1. First character not alphabetic	Error
2. < 1 character	Error
3. 1 character	Acceptable
4. Between 1 and 21 characters	Acceptable
5. 21 characters	Acceptable
6. > 21 characters	Acceptable (truncated to 21 characters)

Equivalence classes for original price of home, current family income, and mortgage balance.

1. < $0.00	Error
2. $0.00	Acceptable
3. $0.01	Acceptable
4. Between $0.01 and $999,999.98	Acceptable
5. $999,999.98	Acceptable
6. $999,999.99	Acceptable
7. $1,000,000.00	Error
8. > $1,000,000.00	Error
9. Character instead of digit	Error (not a number)

Figure 15.13a

Black-Box Test Cases: MSG Foundation (contd)

- Test cases derived from equivalence classes and boundary value analysis (contd)

Equivalence classes for annual property tax and annual homeowner's premium.

1. < $0.00	Error
2. $0.00	Acceptable
3. $0.01	Acceptable
4. Between $0.01 and $99,999.98	Acceptable
5. $99,999.98	Acceptable
6. $99,999.99	Acceptable
7. $100,000.00	Error
8. > $100,000.00	Error
9. Character instead of digit	Error (not a number)

Figure 15.13b

Black-Box Test Cases: MSG Foundation (contd)

- Functional testing test cases

The functions outlined in the specifications document are used to create test cases:

1. Add a mortgage.
2. Add an investment.
3. Modify a mortgage.
4. Modify an investment.
5. Delete a mortgage.
6. Delete an investment.
7. Update operating expenses.
8. Compute funds to purchase houses.
9. Print list of mortgages.
10. Print list of investments.

In addition to these direct tests, it is necessary to perform the following additional tests:

11. Attempt to add a mortgage that is already on file.
12. Attempt to add an investment that is already on file.
13. Attempt to delete a mortgage that is not on file.
14. Attempt to delete an investment that is not on file.
15. Attempt to modify a mortgage that is not on file.
16. Attempt to modify an investment that is not on file.
17. Attempt to delete twice a mortgage that is already on file.
18. Attempt to delete twice an investment that is already on file.
19. Attempt to update each field of a mortgage twice and check that the second version is stored.
20. Attempt to update each field of an investment twice and check that the second version is stored.
21. Attempt to update operating expenses twice and check that second version is stored.

Figure 15.14

15.13 Glass-Box Unit-Testing Techniques

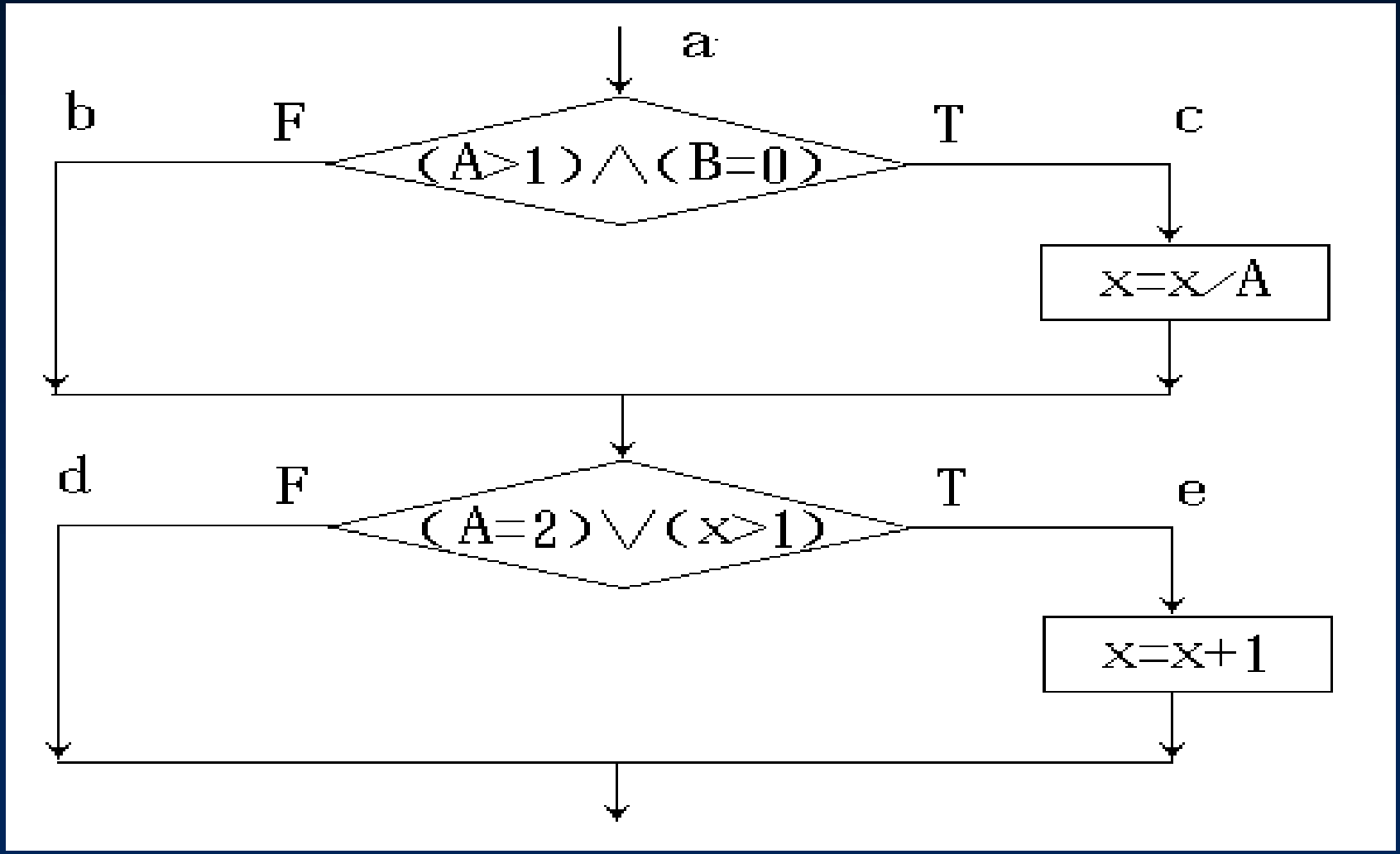
- We will examine
  - Statement coverage
  - Branch coverage
  - Path coverage
  - Linear code sequences
  - All-definition-use path coverage

Test cases in glass-box testing

Logic coverage

- statement coverage
- branch coverage
- condition coverage
- Path coverage

example：L1(a->c->e) ,L2(a->b->d), L3(a->b->e), L4(a->c->d)



$$L1(a \rightarrow c \rightarrow e)$$

$$= \{(A > 1)\ and\ (B = 0)\}\ and\ \{(A = 2)\ or\ (X/A > 1)\}$$

$$= (A > 1)\ and\ (B = 0)\ and\ (A = 2)\ or$$

$$(A > 1)\ and\ (B = 0)\ and\ (X/A > 1)$$

$$= (A = 2)\ and\ (B = 0)\ or$$

$$(A > 1)\ and\ (B = 0)\ and\ (X/A > 1)$$

$$L2\ (a \rightarrow b \rightarrow d)$$

$$= \overline{\{(A > 1)\ and\ (B = 0)\}}\ and\ \overline{\{(A = 2)\ or\ (X > 1)\}}$$

$$= \{\overline{(A > 1)}\ or\ \overline{(B = 0)}\}\ and\ \{\overline{(A = 2)}\ and\ \overline{(X > 1)}\}$$

$$= \overline{(A > 1)}\ and\ \overline{(A = 2)}\ and\ \overline{(X > 1)}\ or$$

$$\overline{(B = 0)}\ and\ \overline{(A = 2)}\ and\ \overline{(X > 1)}$$

$$= (A \leq 1)\ and\ (X \leq 1)\ or$$

$$(B \neq 0)\ and\ (A \neq 2)\ and\ (X \leq 1)$$

$$L3(a \rightarrow b \rightarrow c)$$

$$= \overline{\{(A > 1)\ and\ (B = 0)\}}\ and\ \{(A = 2)\ or\ (X > 1)\}$$

$$= \{\overline{(A > 1)}\ or\ \overline{(B = 0)}\}\ and\ \{(A = 2)\ or\ (X > 1)\}$$

$$= \overline{(A > 1)}\ \text{and}\ (X > 1)\ \text{or}$$

$$\overline{(B = 0)}\ \text{and}\ (A = 2)\ \text{or}\ \overline{(B = 0)}\ \text{and}\ (X > 1)$$

$$= (A \leq 1)\ and\ (X > 1)\ or$$

$$(B \neq 0)\ and\ (A = 2)\ or\ (B \neq 0)\ and\ (X > 1)$$

$$L4\ (a \rightarrow c \rightarrow d)$$

$$= \{(A > 1)\ and\ (B = 0)\}\ and\ \overline{\{(A = 2)\ or\ (X/A > 1)\}}$$

$$= (A > 1)\ and\ (B = 0)\ and\ (A \neq 2)\ and\ (X/A \leq 1)$$

Statement coverage

- The simplest form of glass-box unit testing is **statement coverage**, that is, running a series of test cases during which every statement is executed at least once.

- Format of test case
  【input:(A, B, X), output:(A, B, X)】
- Test case for statement coverage:
  【(2, 0, 4), (2, 0, 3)】
  cover ace【L1】

$$(A = 2)\ and\ (B = 0)\ or$$
$$(A > 1)\ and\ (B = 0)\ and\ (X/A > 1)$$

branch coverage

- An improvement over statement coverage is **branch coverage** , that is, running a series of tests to ensure that all branches are tested at least once.

- 【(2, 0, 4), (2, 0, 3)】cover ace【L1】
  【(1, 1, 1), (1, 1, 1)】cover abd【L2】

$$(A = 2)\ and\ (B = 0)\ or$$

$$(A > 1)\ and\ (B = 0)\ and\ (X/A > 1)$$

$$(A \le 1)\ and\ (X \le 1)\ or$$

$$(B \ne 0)\ and\ (A \ne 2)\ and\ (X \le 1)$$

- If choose L3 and L4, another group of test cases:
  【(2, 1, 1), (2, 1, 2)】cover abe【L3】
  【(3, 0, 3), (3, 1, 1)】cover acd【L4】

$$(A \leq 1)\ and\ (X > 1)\ or\ (B \neq 0)\ and\ (A = 2)\ or\ (B \neq 0)\ and\ (X > 1)$$

$$(A > 1)\ and\ (B = 0)\ and\ (A \neq 2)\ and\ (X/A \leq 1)$$

Condition coverage

- Condition coverage is also known as Predicate Coverage in which each one of the Boolean expression have been evaluated to both TRUE and FALSE。
- We can stamp the values of all decisions,
- Fist decision:
  - if A>1 is true T_1, if false $\overline{T_1}$
    if B=0 is true T_2, is false $\overline{T_2}$

- Second decision:
  - if A=2 is true T_3 is false $\overline{T_3}$
    if X>1 is true T_4, is false $\overline{T_4}$

Test cases	coverage branch	evaluation
【(2, 0, 4), (2, 0, 3)】	L1(c, e)	$T_1T_2T_3T_4$
【(1, 0, 1), (1, 0, 1)】	L2(b, d)	$\overline{T_1}T_2\overline{T_3}T_4$
【(2, 1, 1), (2, 1, 2)】	L3(b, e)	$T_1\overline{T_2}T_3\overline{T_4}$

Branch-condition coverage

- Every point of entry and exit in the program has been invoked at least once, every condition in a decision in the program has taken all possible outcomes at least once, and every decision in the program has taken all possible outcomes at least once.

test cases coverage branch evaluation

【(2, 0, 4), (2, 0, 3)】 L1(a,c, e)

【(1, 1, 1), (1, 1, 1)】 L2(a,b, d)

$T_1T_2T_3T_4$

$\overline{T_1T_2T_3T_4}$

$$(A = 2)\ and\ (B = 0)\ or$$

$$(A > 1)\ and\ (B = 0)\ and\ (X/A > 1)$$

$$(A \le 1)\ and\ (X \le 1)\ or$$

$$(B \ne 0)\ and\ (A \ne 2)\ and\ (X \le 1)$$

Path coverage

- The most powerful form of structural testing is **path coverage** , that is, testing all paths.

test cases	path	coverage condition
【(2, 0, 4), (2, 0, 3)】	ace (L1)	$T_1T_2T_3T_4$
【(1, 1, 1), (1, 1, 1)】	abd (L2)	$\overline{T_1}\overline{T_2}\overline{T_3}T_4$
【(1, 1, 2), (1, 1, 3)】	abe (L3)	$\overline{T_1}\overline{T_2}T_3T_4$
【(3, 0, 3), (3, 0, 1)】	acd (L4)	$T_1T_2\overline{T_3}\overline{T_4}$

- *Statement coverage*:
  - Running a set of test cases in which every statement is executed at least once
  - A CASE tool needed to keep track
- Weakness
  - Branch statements
- Both statements can be executed without the fault showing up

```
if (s > 1 && t == 0)
   x = 9;

Test case:          s = 2, t = 0.
```

Figure 15.15

Structural Testing: Branch Coverage

- Running a set of test cases in which every branch is executed at least once (as well as all statements)
  - This solves the problem on the previous slide
  - Again, a CASE tool is needed

Structural Testing: Path Coverage

- Running a set of test cases in which every path is executed at least once (as well as all statements)
- Problem:
  - The number of paths may be very large
- We want a weaker condition than all paths but that shows up more faults than branch coverage

Linear Code Sequences

- Identify the set of points L from which control flow may jump, plus entry and exit points
- Restrict test cases to paths that begin and end with elements of L
- This uncovers many faults without testing every path

All-Definition-Use-Path Coverage

- Each occurrence of variable, `zz` say, is labeled either as
  - The *definition* of a variable

    `zz = 1` or `read (zz)`
  - or the *use* of variable

    `y = zz + 3` or `if (zz < 9) errorB ()`
- Identify all paths from the definition of a variable to the use of that definition
  - This can be done by an automatic tool
- A test case is set up for each such path

All-Definition-Use-Path Coverage (contd)

- Disadvantage:
  - Upper bound on number of paths is 2^d, where d is the number of branches
- In practice:
  - The actual number of paths is proportional to d
- This is therefore a practical test case selection technique

Infeasible Code

- It may not be possible to test a specific statement
  - We may have an infeasible path ("dead code") in the artifact

- Frequently this is evidence of a fault

```
if (k < 2)
{
   if (k > 3)              [should be k > -3]
          ↑
      x = x * k;
}
                    (a)

for (j = 0; j < 0; j++)   [should be j < 10]
                ↑
   total = total + value[j];
                    (b)
```

Figure 15.16

15.13.2 Complexity Metrics

- A quality assurance approach to glass-box testing
- Artifact `m1` is more “complex” than artifact `m2`
  - Intuitively, `m1` is more likely to have faults than artifact `m2`
- If the complexity is unreasonably high, redesign and then reimplement that code artifact
  - This is cheaper and faster than trying to debug a fault-prone code artifact

Lines of Code

- The simplest measure of complexity
  - Underlying assumption: There is a constant probability p that a line of code contains a fault
- Example
  - The tester believes each line of code has a 2% chance of containing a fault.
  - If the artifact under test is 100 lines long, then it is expected to contain 2 faults
- The number of faults is indeed related to the size of the product as a whole

Other Measures of Complexity

- Cyclomatic complexity *M* (McCabe)
  - Essentially the number of decisions (branches) in the artifact
  - Easy to compute
  - A surprisingly good measure of faults (but see next slide)
- In one experiment, artifacts with $M > 10$ were shown to have statistically more errors

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

15.15 Comparison of Unit-Testing Techniques

- Experiments comparing
  - Black-box testing
  - Glass-box testing
  - Reviews
- [Myers, 1978] 59 highly experienced programmers
  - All three methods were equally effective in finding faults
  - Code inspections were less cost-effective
- [Hwang, 1981]
  - All three methods were equally effective

Comparison of Unit-Testing Techniques (contd)

- [Basili and Selby, 1987] 42 advanced students in two groups, 32 professional programmers
- Advanced students, group 1
  - No significant difference between the three methods
- Advanced students, group 2
  - Code reading and black-box testing were equally good
  - Both outperformed glass-box testing
- Professional programmers
  - Code reading detected more faults
  - Code reading had a faster fault detection rate

Comparison of Unit-Testing Techniques (contd)

- Conclusion
  - Code inspection is at least as successful at detecting faults as glass-box and black-box testing

Cleanroom

- A different approach to software development
- Incorporates
  - An incremental process model
  - Formal techniques
  - Reviews

Cleanroom (contd)

- Prototype automated documentation system for the U.S. Naval Underwater Systems Center
- 1820 lines of FoxBASE
  - 18 faults were detected by “functional verification”
  - Informal proofs were used
  - 19 faults were detected in walkthroughs before compilation
  - There were NO compilation errors
  - There were NO execution-time failures

Cleanroom (contd)

- *Testing fault rate* counting procedures differ:
- Usual paradigms:
  - Count faults after informal testing is complete (once SQA starts)
- Cleanroom
  - Count faults after inspections are complete (once compilation starts)

Report on 17 Cleanroom Products

- Operating system
  - 350,000 LOC
  - Developed in only 18 months
  - By a team of 70
  - The testing fault rate was only 1.0 faults per KLOC
- Various products totaling 1 million LOC
  - Weighted average testing fault rate: 2.3 faults per KLOC
- “[R]emarkable quality achievement”

Potential Problems When Testing Objects

- We must inspect classes and objects
- We can run test cases on objects (but not on classes)

Potential Problems When Testing Obj. (contd)

- A typical classical module:
  - About 50 executable statements
  - Give the input arguments, check the output arguments
- A typical object:
  - About 30 methods, some with only 2 or 3 statements
  - A method often does not return a value to the caller — it changes state instead
  - It may not be possible to check the state because of information hiding
  - Example: Method `determineBalance` — we need to know `accountBalance` before, after

Potential Problems When Testing Obj. (contd)

- We need additional methods to return values of all state variables
  - They must be part of the test plan
  - Conditional compilation may have to be used
- An inherited method may still have to be tested (see next four slides)

Potential Problems When Testing Obj. (contd)

- Java implementation of a tree hierarchy

```
class RootedTreeClass
{
   …
   void displayNodeContents (Node a);
   void printRoutine (Node b);
//
// method displayNodeContents uses method printRoutine
//
   …
}

class BinaryTreeClass extends RootedTreeClass
{
   …
   void displayNodeContents (Node a);
//
// method displayNodeContents defined in this class uses
// method printRoutine inherited from ClassRootedTree
//
   …
}

class BalancedBinaryTreeClass extends BinaryTreeClass
{
   …
   void printRoutine (Node b);
//
// method displayNodeContents (inherited from BinaryTreeClass) uses this
// local version of printRoutine within class BalancedBinaryTreeClass
//
   …
}
```

Figure 15.17

Potential Problems When Testing Obj. (contd)

- Top half

```
class RootedTreeClass
{
   …
   void displayNodeContents (Node a);
   void printRoutine (Node b);
//
// method displayNodeContents uses method printRoutine
//
   …
}

class BinaryTreeClass extends RootedTreeClass
{
   …
   void displayNodeContents (Node a);
//
// method displayNodeContents defined in this class uses
// method printRoutine inherited from ClassRootedTree
//
   …
}
```

Figure 15.17
(top half)

- When `displayNodeContents` is invoked in **`BinaryTreeClass`**, it uses **`RootedTreeClass`**`.printRoutine`

Potential Problems When Testing Obj. (contd)

- Bottom half

```
class BinaryTreeClass extends RootedTreeClass
{
   ...
   void displayNodeContents (Node a);
//
// method displayNodeContents defined in this class uses
// method printRoutine inherited from ClassRootedTree
//
   ...
}

class BalancedBinaryTreeClass extends BinaryTreeClass
{
   ...
   void printRoutine (Node b);
//
// method displayNodeContents (inherited from BinaryTreeClass) uses this
// local version of printRoutine within class BalancedBinaryTreeClass
//
   ...
}
```

Figure 15.17 (bottom half)

- When `displayNodeContents` is invoked in **`BalancedBinaryTreeClass`**, it uses **`BalancedBinaryTreeClass`**`.printRoutine`

Potential Problems When Testing Obj. (contd)

- Bad news
  - **`BinaryTreeClass`**`.displayNodeContents` must be retested from scratch when reused in **`BalancedBinaryTreeClass`**
  - It invokes a different version of `printRoutine`
- Worse news
  - For theoretical reasons, we need to test using totally different test cases

Potential Problems When Testing Obj. (contd)

- Making state variables visible
  - Minor issue
- Retesting before reuse
  - Arises only when methods interact
  - We can determine when this retesting is needed
- These are not reasons to abandon the object-oriented paradigm

15.18 Management Aspects of Unit Testing

- We need to know when to stop testing
- A number of different techniques can be used
  - Cost–benefit analysis
  - Risk analysis
  - Statistical techniques

15.19 When to Rewrite Rather Than Debug

- When a code artifact has too many faults
  - It is cheaper to redesign, then recode

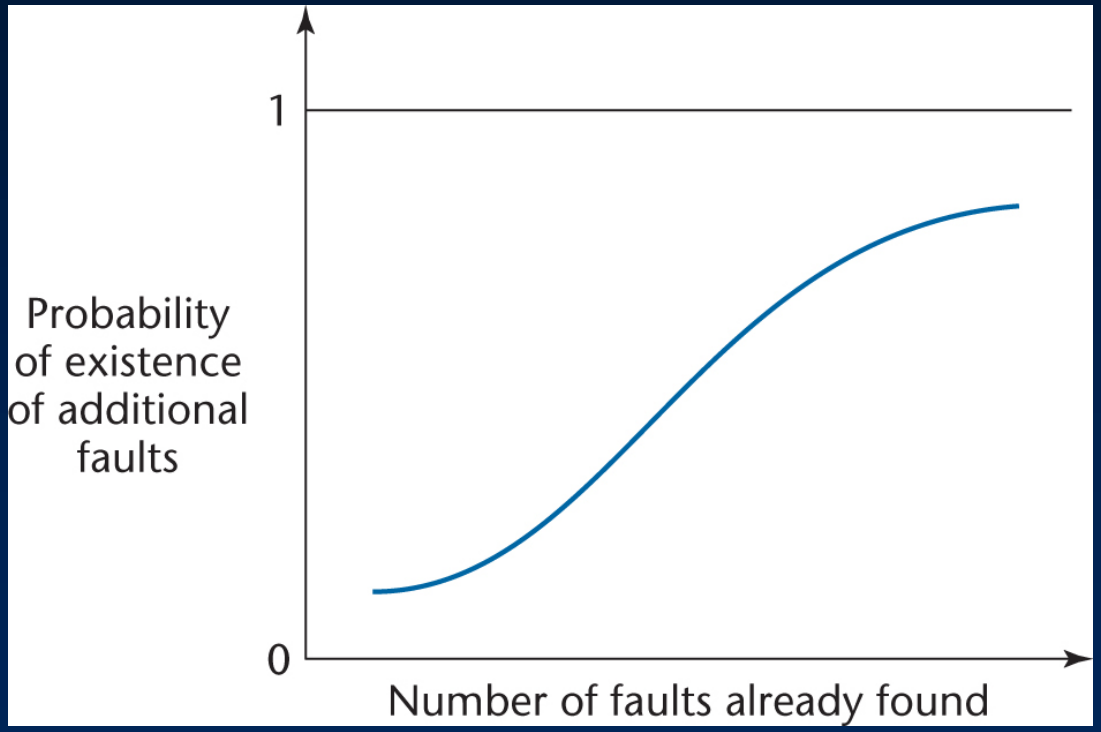


Figure 15.18

- The risk and cost of further faults are too great

Fault Distribution in Modules Is Not Uniform

- [Myers, 1979]
  - 47% of the faults in OS/370 were in only 4% of the modules
- [Endres, 1975]
  - 512 faults in 202 modules of DOS/VS (Release 28)
  - 112 of the modules had only one fault
  - There were modules with 14, 15, 19 and 28 faults, respectively
  - The latter three were the largest modules in the product, with over 3000 lines of DOS macro assembler language
  - The module with 14 faults was relatively small, and very unstable
  - A prime candidate for discarding, redesigning, recoding

When to Rewrite Rather Than Debug (contd)

- For every artifact, management must predetermine the maximum allowed number of faults during testing
- If this number is reached
  - Discard
  - Redesign
  - Recode
- The maximum number of faults allowed *after delivery* is ZERO

15.20 Integration Testing

- The testing of each new code artifact when it is added to what has already been tested
- Special issues can arise when testing graphical user interfaces — see next slide

Integration Testing of Graphical User Interfaces

- GUI test cases include
  - Mouse clicks, and
  - Key presses
- These types of test cases cannot be stored in the usual way
  - We need special CASE tools
- Examples:
  - QAPartner
  - XRunner

15.21 Product Testing

- Product testing for COTS software
  - Alpha, beta testing
- Product testing for custom software
  - The SQA group must ensure that the product passes the acceptance test
  - Failing an acceptance test has bad consequences for the development organization

Product Testing for Custom Software

- The SQA team must try to approximate the acceptance test
  - Black box test cases for the product as a whole
  - Robustness of product as a whole
    - » *Stress testing* (under peak load)
    - » *Volume testing* (e.g., can it handle large input files?)
  - All constraints must be checked
  - All documentation must be
    - » Checked for correctness
    - » Checked for conformity with standards
    - » Verified against the current version of the product

Product Testing for Custom Software (contd)

- The product (code plus documentation) is now handed over to the client organization for acceptance testing

15. 22 Acceptance Testing

- The client determines whether the product satisfies its specifications
- Acceptance testing is performed by
  - The client organization, or
  - The SQA team in the presence of client representatives, or
  - An independent SQA team hired by the client

Acceptance Testing (contd)

- The four major components of acceptance testing are
  - Correctness
  - Robustness
  - Performance
  - Documentation

- These are precisely what was tested by the developer during product testing

Acceptance Testing (contd)

- The key difference between product testing and acceptance testing is
  - Acceptance testing is performed on actual data
  - Product testing is preformed on test data, which can never be real, by definition

15.23 The Test Workflow: The MSG Foundation Case Study

- The C++ and Java implementations were tested against
  - The black-box test cases of Figures 15.13 and 15.14, and
  - The glass-box test cases of Problems 15.30 through 15.34

15.24 CASE Tools for Implementation

- CASE tools for implementation of code artifacts were described in Chapter 5

- CASE tools for integration include
  - Version-control tools, configuration-control tools, and build tools
  - Examples:
    - » *rcs*, *sccs*, PCVS, SourceSafe, Subversion

CASE Tools for Implementation (contd)

- Configuration-control tools
  - Commercial
    - » PCVS, SourceSafe
  - Open source
    - » CVS

15.24.1 CASE Tools for the Complete Software Process

- A large organization needs an environment
- A medium-sized organization can probably manage with a workbench
- A small organization can usually manage with just tools

15.24.2 Integrated Development Environments

- The usual meaning of “integrated”
  - User interface integration
  - Similar “look and feel”
  - Most successful on the Macintosh
- There are also other types of integration
- Tool integration
  - All tools communicate using the same format
  - Example:
    - » Unix Programmer’s Workbench

Process Integration

- The environment supports one specific process
- Subset: Technique-based environment
  - Formerly: “method-based environment”
  - Supports a specific technique, rather than a complete process
  - Environments exist for techniques like
    - » Structured systems analysis
    - » Petri nets

Technique-Based Environment

- Usually comprises
  - Graphical support for analysis, design
  - A data dictionary
  - Some consistency checking
  - Some management support
  - Support and formalization of manual processes
  - Examples:
    - » Analyst/Designer
    - » Software through Pictures
    - » IBM Rational Rose
    - » Rhapsody (for Statecharts)

Technique-Based Environments (contd)

- Advantage of a technique-based environment
  - The user is forced to use one specific method, correctly
- Disadvantages of a technique-based environment
  - The user is forced to use one specific method, so that the method must be part of the software process of that organization

15.24.3 Environments for Business Application

- The emphasis is on ease of use, including
  - A user-friendly GUI generator,
  - Standard screens for input and output, and
  - A code generator
    - » Detailed design is the lowest level of abstraction
    - » The detailed design is the input to the code generator
- Use of this “programming language” should lead to a rise in productivity
- Example:
  - Oracle Development Suite

15.24.4 Public Tool Infrastructure

- PCTE — Portable common tool environment
  - *Not* an environment
  - An infrastructure for supporting CASE tools (similar to the way an operating system provides services for user products)
  - Adopted by ECMA (European Computer Manufacturers Association)

- Example implementations:
  - IBM, Emeraude

15.24.5 Potential Problems with Environments

- No one environment is ideal for all organizations
  - Each has its strengths and its weaknesses
- Warning 1
  - Choosing the wrong environment can be worse than no environment
  - Enforcing a wrong technique is counterproductive
- Warning 2
  - Shun CASE environments below CMM level 3
  - We cannot automate a nonexistent process
  - However, a CASE tool or a CASE workbench is fine

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

Overview of the MSG Foundation Case Study

Implementation workflow	Section 15.8, Appendix H, Appendix I
Black-box test cases	Section 15.12
Test workflow	Section 15.23

Figure 15.19