

Slide 15.1

Object-Oriented and Classical Software Engineering

Eighth Edition, WCB/McGraw-Hill, 2011

Stephen R. Schach

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

CHAPTER 15

Slide 15.2

IMPLEMENTATION

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Overview

Slide 15.3

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Overview (contd)

Slide 15.4

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Overview (contd)

Slide 15.5

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Implementation

Slide 15.6

- Real-life products are generally too large to be implemented by a single programmer
- This chapter therefore deals with programming-in-the-many

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.1 Choice of Programming Language (contd)

Slide 15.7

- 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?

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Choice of Programming Language (contd)

Slide 15.8

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Choice of Programming Language (contd)

Slide 15.9

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Choice of Programming Language (contd)

Slide 15.10

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Choice of Programming Language (contd)

Slide 15.11

- 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)?

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.2 Fourth Generation Languages

Slide 15.12

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Fourth Generation Languages (contd)

Slide 15.13

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Fourth Generation Languages (contd)

Slide 15.14

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Fourth Generation Languages (contd)

Slide 15.15

- Example
 - See Just in Case You Wanted to Know Box 15.2
- The power of a nonprocedural language, and the price

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Productivity Increases with a 4GL?

Slide 15.16

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Actual Experiences with 4GLs

Slide 15.17

- ▣ 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Actual Experiences with 4GLs (contd)

Slide 15.18

- ▣ 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Fourth Generation Languages (contd)

Slide 15.19

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Dangers of a 4GL

Slide 15.20

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Dangers of a 4GL (contd)

Slide 15.21

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.3 Good Programming Practice

Slide 15.22

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.3.1 Use of Consistent and Meaningful Variable Names

Slide 15.23

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Consistent and Meaningful Variable Names

Slide 15.24

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.3.2 The Issue of Self-Documenting Code

Slide 15.25

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Self-Documenting Code Example

Slide 15.26

- 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?

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Prologue Comments

Slide 15.27

□ 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Other Comments

Slide 15.28

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.3.3 Use of Parameters

Slide 15.29

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.3.4 Code Layout for Increased Readability

Slide 15.30

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.3.5 Nested `if` Statements

Slide 15.31

□ 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

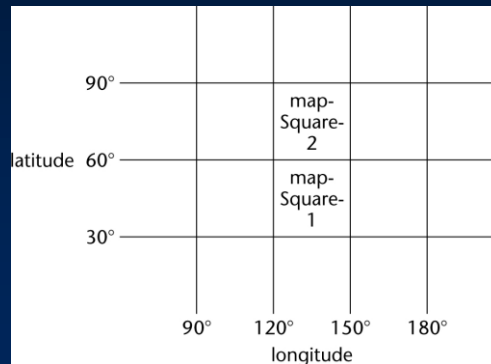


Figure 15.2

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Nested `if` Statements (contd)

Slide 15.32

□ 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Nested `if` Statements (contd)

Slide 15.33

□ 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Nested `if` Statements (contd)

Slide 15.34

□ 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Nested `if` Statements (contd)

Slide 15.35

- 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>
```

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Nested `if` Statements (contd)

Slide 15.36

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.4 Programming Standards

Slide 15.37

- 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”

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Remarks on Programming Standards

Slide 15.38

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Examples of Good Programming Standards

Slide 15.39

- “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”

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Remarks on Programming Standards (contd)

Slide 15.40

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.5 Code Reuse

Slide 15.41

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.6 Integration

Slide 15.42

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Product with 13 Modules

Slide 15.43

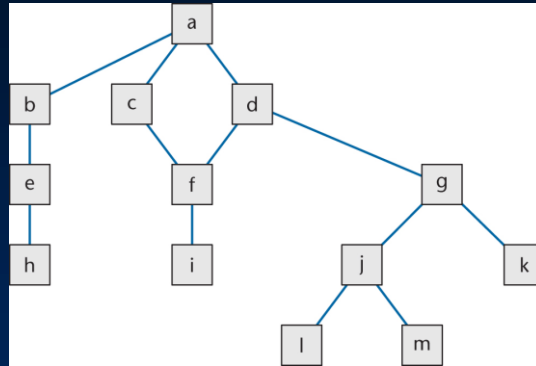


Figure 15.6

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Implementation, Then Integration

Slide 15.44

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Drivers and Stubs

Slide 15.45

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Implementation, Then Integration (contd)

Slide 15.46

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Implementation, Then Integration (contd)

Slide 15.47

- Solution to both problems
 - Combine unit and integration testing

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.6.1 Top-down Integration

Slide 15.48

- If code artifact m_{Above} sends a message to artifact m_{Below} , then m_{Above} is implemented and integrated before m_{Below}
- One possible top-down ordering is
 - a, b, c, d, e, f, g, h, i, j, k, l, m

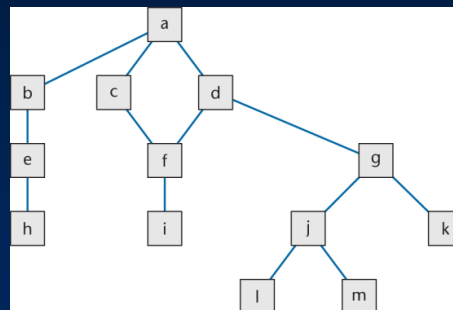


Figure 15.6 (again)

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Top-down Integration (contd)

Slide 15.49

- Another possible top-down ordering is

```

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

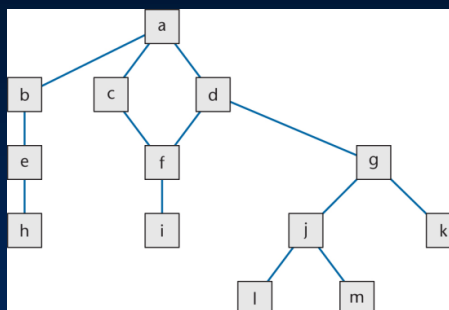


Figure 15.6 (again)

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Top-down Integration (contd)

Slide 15.50

- Advantage 1: Fault isolation
 - A previously successful test case fails when m_{New} is added to what has been tested so far
 - » The fault must lie in m_{New} or the interface(s) between m_{New} 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Top-down Integration (contd)

Slide 15.51

- ▣ 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Top-down Integration (contd)

Slide 15.52

- ▣ 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.6.2 Bottom-up Integration

Slide 15.53

- If code artifact m_{Above} calls code artifact m_{Below} , then m_{Below} is implemented and integrated before m_{Above}

- One possible bottom-up ordering is

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

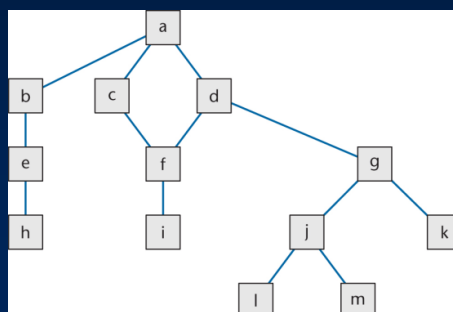


Figure 15.6 (again)

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.6.2 Bottom-up Integration

Slide 15.54

- Another possible bottom-up ordering is

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

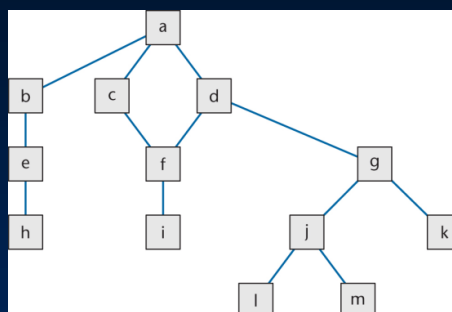


Figure 15.6 (again)

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Bottom-up Integration (contd)

Slide 15.55

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Bottom-up Integration (contd)

Slide 15.56

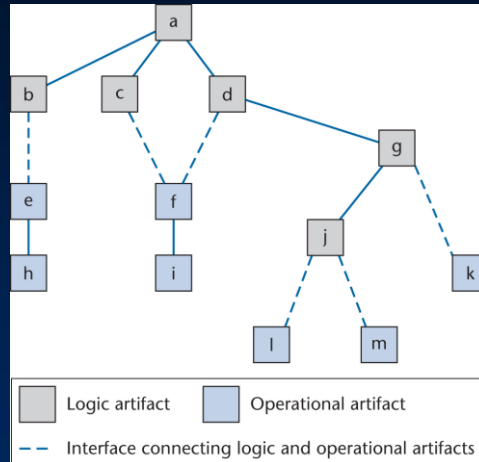
- Difficulty 1
 - Major design faults are detected late
- Solution
 - Combine top-down and bottom-up strategies making use of their strengths and minimizing their weaknesses

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.6.3 Sandwich Integration

Slide 15.57

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



Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Sandwich Integration (contd)

Slide 15.58

- ❑ 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Summary

Slide 15.59

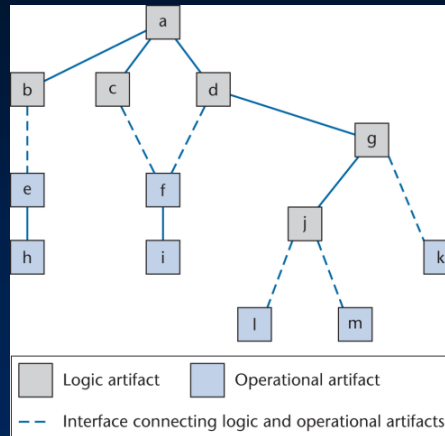


Figure 15.8

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.6.4 Integration of Object-Oriented Products

Slide 15.60

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.6.5 Management of Integration

Slide 15.61

- Example:
 - Design document used by programmer P_1 (who coded code object o_1) shows o_1 sends a message to o_2 passing 4 arguments
 - Design document used by programmer P_2 (who coded code artifact o_2) states clearly that only 3 arguments are passed to o_2
- Solution:
 - The integration process must be run by the SQA group
 - They have the most to lose if something goes wrong

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.7 The Implementation Workflow

Slide 15.62

- 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*

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

The Implementation Workflow (contd)

Slide 15.63

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.8 The Implementation Workflow: The MSG Foundation Case Study

Slide 15.64

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.9 The Test Workflow: Implementation

Slide 15.65

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.10 Test Case Selection

Slide 15.66

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.10.1 Testing to Specifications versus Testing to Code

Slide 15.67

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.10.2 Feasibility of Testing to Specifications

Slide 15.68

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Feasibility of Testing to Specifications (contd)

Slide 15.69

- Suppose the specifications include 20 factors, each taking on 4 values
 - There are 4^{20} or 1.1×10^{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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.10.3 Feasibility of Testing to Code

Slide 15.70

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Feasibility of Testing to Code (contd)

Slide 15.71

□ 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Feasibility of Testing to Code (contd)

Slide 15.72

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

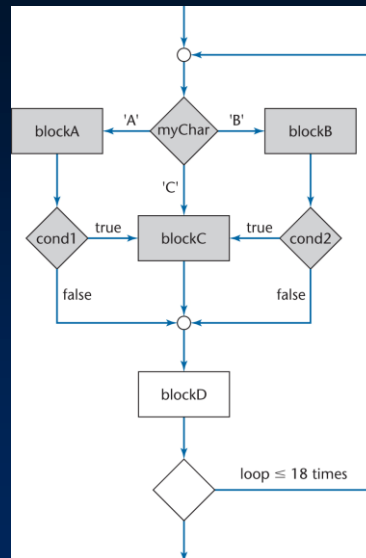


Figure 15.10

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Feasibility of Testing to Code (contd)

Slide 15.73

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Feasibility of Testing to Code (contd)

Slide 15.74

- A path can be tested only if it is present

```

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

x = n/d;
    (b)
  
```

Figure 15.12

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Feasibility of Testing to Code (contd)

Slide 15.75

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.11 Black-Box Unit-testing Techniques

Slide 15.76

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Black-Box Unit-testing Techniques (contd)

Slide 15.77

- 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)

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.11.1 Equivalence Testing and Boundary Value Analysis

Slide 15.78

- 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*

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Equivalence Testing

Slide 15.79

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Boundary Value Analysis

Slide 15.80

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Database Example (contd)

Slide 15.81

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Database Example (contd)

Slide 15.82

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Equivalence Testing of Output Specifications

Slide 15.83

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Overall Strategy

Slide 15.84

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.11.2 Functional Testing

Slide 15.85

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Functional Testing (contd)

Slide 15.86

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Functional Testing (contd)

Slide 15.87

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

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

Slide 15.88

- 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 mortgages	
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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Black-Box Test Cases: MSG Foundation (contd)

Slide 15.89

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Black-Box Test Cases: MSG Foundation (contd)

Slide 15.90

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.13 Glass-Box Unit-Testing Techniques

Slide 15.91

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.13.1 Structural Testing: Statement, Branch, and Path Coverage

Slide 15.92

- *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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Structural Testing: Branch Coverage

Slide 15.93

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Structural Testing: Path Coverage

Slide 15.94

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Linear Code Sequences

Slide 15.95

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

All-Definition-Use-Path Coverage

Slide 15.96

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

All-Definition-Use-Path Coverage (contd)

Slide 15.97

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Infeasible Code

Slide 15.98

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.13.2 Complexity Metrics

Slide 15.99

- A quality assurance approach to glass-box testing
- Artifact m_1 is more “complex” than artifact m_2
 - Intuitively, m_1 is more likely to have faults than artifact m_2
- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Lines of Code

Slide 15.100

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Other Measures of Complexity

Slide 15.101

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Problem with Complexity Metrics

Slide 15.102

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Code Walkthroughs and Inspections

Slide 15.103

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.15 Comparison of Unit-Testing Techniques

Slide 15.104

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Comparison of Unit-Testing Techniques (contd)

Slide 15.105

- [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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Comparison of Unit-Testing Techniques (contd)

Slide 15.106

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Cleanroom

Slide 15.107

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Cleanroom (contd)

Slide 15.108

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Cleanroom (contd)

Slide 15.109

- *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)

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Report on 17 Cleanroom Products

Slide 15.110

- 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”

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Potential Problems When Testing Objects

Slide 15.111

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Potential Problems When Testing Obj. (contd)

Slide 15.112

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Potential Problems When Testing Obj. (contd)

Slide 15.113

- 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)

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Potential Problems When Testing Obj. (contd)

Slide 15.114

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Potential Problems When Testing Obj. (contd)

Slide 15.115

□ 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`

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Potential Problems When Testing Obj. (contd)

Slide 15.116

□ 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`

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Potential Problems When Testing Obj. (contd)

Slide 15.117

❑ 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Potential Problems When Testing Obj. (contd)

Slide 15.118

❑ 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.18 Management Aspects of Unit Testing

Slide 15.119

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.19 When to Rewrite Rather Than Debug

Slide 15.120

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

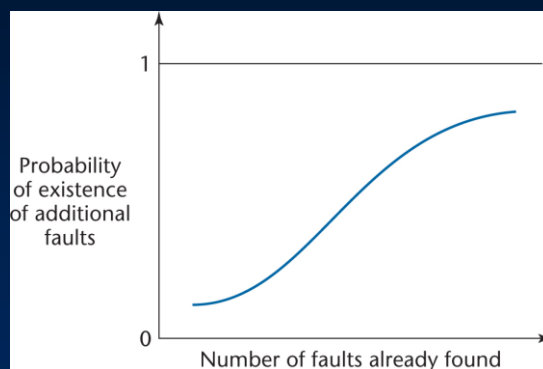


Figure 15.18

- The risk and cost of further faults are too great

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Fault Distribution in Modules Is Not Uniform

Slide 15.121

- [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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

When to Rewrite Rather Than Debug (contd)

Slide 15.122

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.20 Integration Testing

Slide 15.123

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Integration Testing of Graphical User Interfaces

Slide 15.124

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.21 Product Testing

Slide 15.125

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Product Testing for Custom Software

Slide 15.126

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Product Testing for Custom Software (contd)

Slide 15.127

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15. 22 Acceptance Testing

Slide 15.128

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Acceptance Testing (contd)

Slide 15.129

- The four major components of acceptance testing are
 - Correctness
 - Robustness
 - Performance
 - Documentation
- These are precisely what was tested by the developer during product testing

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Acceptance Testing (contd)

Slide 15.130

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.23 The Test Workflow: The MSG Foundation Case Study

Slide 15.131

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.24 CASE Tools for Implementation

Slide 15.132

- 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:
 - » *rscs*, *sccs*, PCVS, SourceSafe, Subversion

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

CASE Tools for Implementation (contd)

Slide 15.133

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.24.1 CASE Tools for the Complete Software Process

Slide 15.134

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.24.2 Integrated Development Environments

Slide 15.135

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Process Integration

Slide 15.136

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Technique-Based Environment

Slide 15.137

- 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)

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Technique-Based Environments (contd)

Slide 15.138

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.24.3 Environments for Business Application

Slide 15.139

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.24.4 Public Tool Infrastructure

Slide 15.140

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.24.5 Potential Problems with Environments

Slide 15.141

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.25 Metrics for the Implementation Workflow

Slide 15.142

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

15.26 Challenges of the Implementation Workflow

Slide 15.143

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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Challenges of the Implementation Workflow (contd)

Slide 15.144

- ▣ 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Challenges of the Implementation Phase (contd)

Slide 15.145

- 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

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.

Overview of the MSG Foundation Case Study

Slide 15.146

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

Figure 15.19

Copyright © 2011 by The McGraw-Hill Companies, Inc. All rights reserved.