CS 123
Introduction to Software Engineering

07: Software Implementation

DISCS

SY 2013 - 2014

Overview

- Choice of programming language
- Fourth generation languages
- Good programming practice
- Coding standards
- Module test case selection
- Black-box module-testing techniques
- Glass-box module-testing techniques

Overview (contd)

- Code walkthroughs and inspections
- Comparison of module-testing techniques
- Cleanroom
- Potential problems when testing objects
- Management aspects of module testing
- When to rewrite rather than debug a module
- CASE tools for the implementation phase
- Air Gourmet Case Study: Black-box test cases
- Challenges of the implementation phase

Implementation Phase

- Programming-in-the-many
- Choice of Programming Language
  - Language is usually specified in contract
- But what if the contract specifies
  - 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 "most suitable" programming language?
- Obviously COBOL

Choice of Programming Language (contd)

- What happens when new language (C++, say) is introduced
  - New hires
  - Retrain existing professionals
  - Future products in C++
  - Maintain existing COBOL products
  - Two classes of programmers
    - COBOL maintainers (despised)
    - C++ developers (paid more)
  - Need expensive software, and hardware to run it
  - 100s of person-years of expertise with COBOL wasted

Choice of Programming Language (contd)

- 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 DP applications
- How to choose a programming language
  - Cost-benefit analysis
  - Compute costs, benefits of all relevant languages

Choice of Programming Language (contd)

- Which is the most appropriate object-oriented language?
  - C++ is (unfortunately) C-like
  - 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)?

Fourth Generation Languages

- First generation languages
  - Machine languages
- Second generation languages
  - Assemblers
- Third generation languages
  - High-level languages (COBOL, FORTRAN, C++)
- Fourth generation languages (4GLs)
  - One 3GL statement is equivalent to 5–10 assembler statements
  - Each 4GL statement 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
  - Applications easy, 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

Actual Experiences with 4GLs

- 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 (contd)

- Attitudes of 43 Organizations to 4GLs
  - Use of 4GL reduced users' frustrations
  - Quicker response from DP department
  - 4GLs 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
- Reason
  - No one 4GL has all the necessary features
- Conclusion
  - Care has to be taken in selecting the appropriate 4GL

Additional Note when Using a 4GL

- Dangers of a 4GL
  - Deceptive simplicity
  - End-user programming

Good Programming Practice

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

Good Programming Practice Example

- Module contains variables freqAverage, frequencyMaximum, minFr, frqncyTotl
- Maintenance programmer has to know if freq, frequency, fr, frqncy all refer to the same thing
  - If so, use identical word, preferably frequency, perhaps freq or frqncy, *not* fr
  - If not, use different word (e.g., rate) for different quantity
- Can use frequencyAverage, frequencyMyaximum, frequencyMinimum, frequencyTotal
- Can also use averageFrequency, maximumFrequency, minimumFrequency, totalFrequency
- All four names must come from the same set

Good Programming Practice (contd)

- Issue of self-documenting code
  - Exceedingly rare
- Key issue: Can module be understood easily and unambiguously by
  - SQA team
  - Maintenance programmers
  - All others who have to read the code

Good Programming Practice (contd)

- Example
  - Variable xCoordinateOfPositionOfRobotArm
  - Abbreviated to xCoord
  - Entire module deals with the movement of the robot arm
  - But does the maintenance programmer know this?

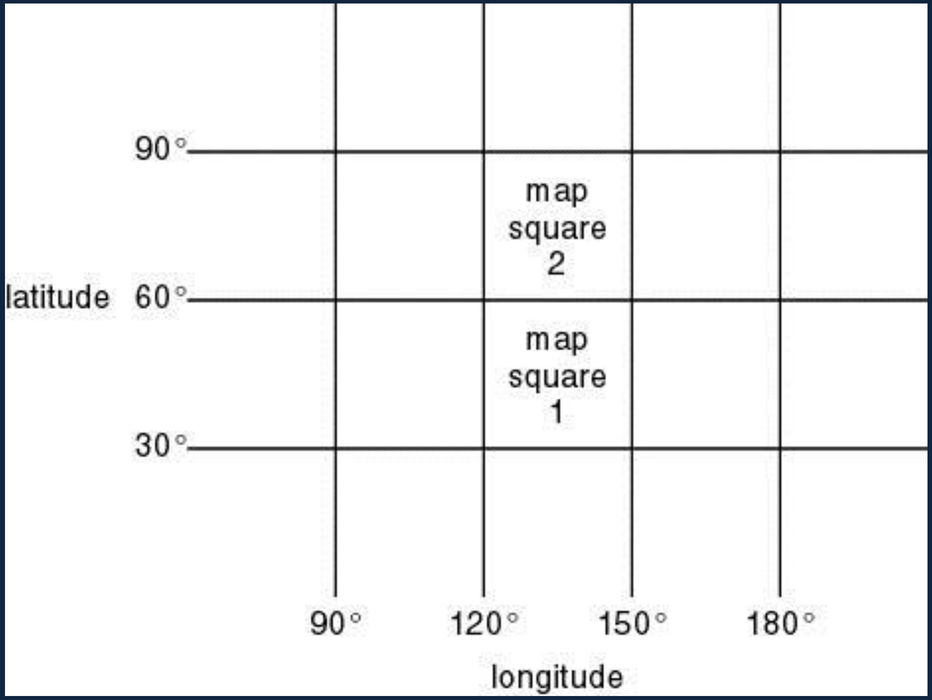
Prologue Comments

- Mandatory at top of every single module
  - Minimum information
    - Module name
    - Brief description of what the module does
    - Programmer's name
    - Date module was coded
    - Date module was approved, and by whom
    - Module parameters
    - Variable names, in alphabetical order, and uses
    - Files accessed by this module
    - Files updated by this module
    - Module i/o
    - Error handling capabilities
    - Name of file of test data (for regression testing)
    - List of modifications made, when, approved by whom
    - Known faults, if any

Other Comments

- Suggestion
  – Comments are essential whenever 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 maintenance programmers
- Code layout for increased readability
  – Use indentation
  – Better, use a pretty-printer
  – Use blank lines

Nested if Statements



- Example
  - Map consists of two squares. Write code to determine whether a point on the Earth's surface lies in map square 1 or map square 2, or is not on the map

Nested if Statements (contd)

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";

- Solution 1. Badly formatted

Nested if Statements (contd)

```
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";
```

- Solution 2. Well-formatted, badly constructed

Nested if Statements (contd)

```
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";
```

- Solution 3. Acceptably nested

Nested **if** Statements (contd)

- 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>*

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

Programming 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

Remarks on Programming Standards (contd)

- Examples of good programming standards
  - "Nesting of **if** statements should not exceed a depth of 3, except with prior approval from the team leader"
  - "Modules should consist of between 35 and 50 statements, except with prior approval from the team leader"
  - "Use of gotos should be avoided. However, with prior approval from the team leader, a forward goto may be used for error handling"

Remarks on Programming Standards (contd)

- Aim of standards is to make maintenance easier
  - If it makes development difficult, then must be modified
  - Overly restrictive standards are counterproductive
  - Quality of software suffers

Software Quality Control

- After preliminary testing by the programmer, the module is handed over to the SQA group

Module Reuse

- The most common form of reuse

Module Test Case Selection

- Worst way—random testing
- Need systematic way to construct test cases

Module Test Case Selection (contd)

- Two extremes to testing

1. Test to specifications (also called black-box, data-driven, functional, or input/output driven testing)
   – Ignore code. Use specifications to select test cases
2. Test to code (also called glass-box, logic-driven, structured, or path-oriented testing)
   – Ignore specifications. Use code to select test cases

Feasibility of Testing to Specifications

- Example
  - Specifications for data processing product include 5 types of commission and 7 types of discount
  - 35 test cases
- Cannot say that commission and discount are computed in two entirely separate modules—the structure is irrelevant

Feasibility of Testing to Specifications

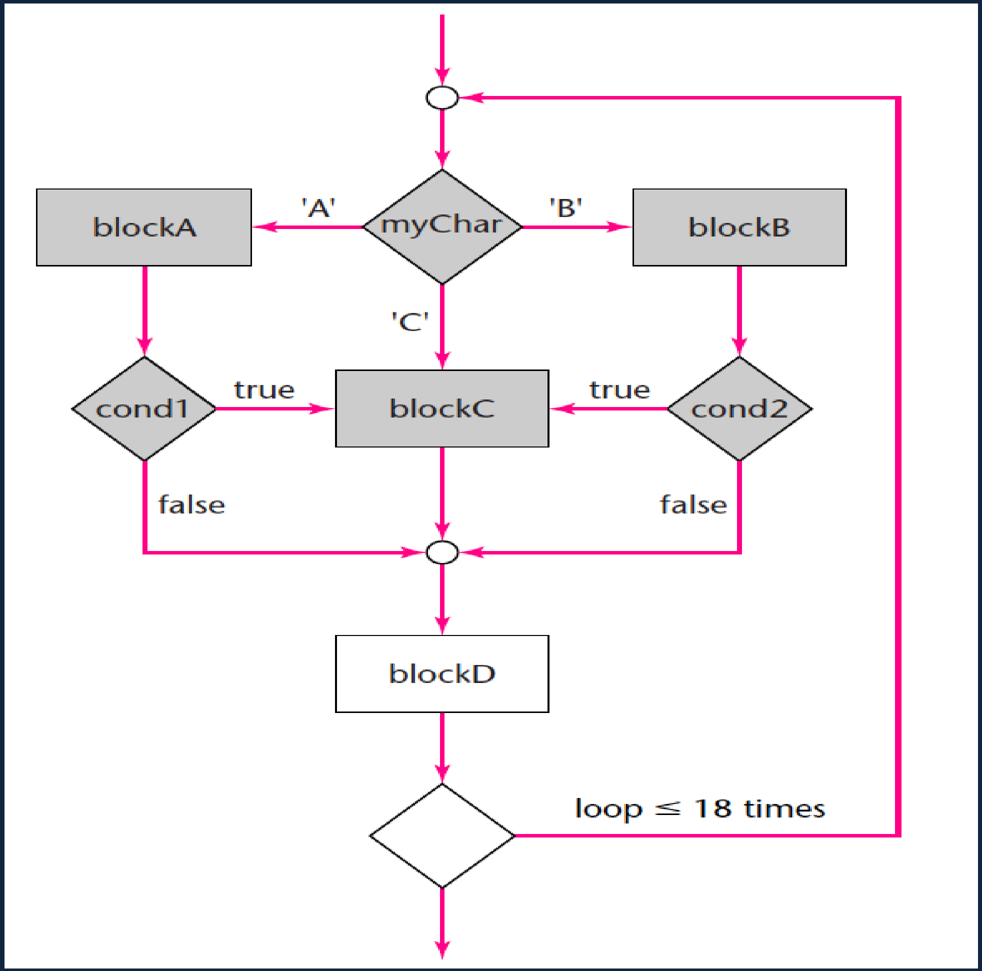
- Suppose specs include 20 factors, each taking on 4 values
  - 4^{20} or 1.1×10^{12} test cases
  - If each takes 30 seconds to run, running all test cases takes > 1 million years
- Combinatorial explosion makes testing to specifications impossible

Feasibility of Testing to Code

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

```
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;
}
```

Feasibility of Testing to Code (contd)



- Flowchart has over 10^{12} different paths

Feasibility of Testing to Code (contd)

```
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: $x = y = z = 2$

- Can exercise every path without detecting every fault

Feasibility of Testing to Code (contd)

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

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

- Path can be tested only if it is present

Coping with the Combinatorial Explosion

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

Coping with the Combinatorial Explosion

- 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)

Black-Box Module Testing Methods

- Equivalence Testing
- Example
  - Specifications for DBMS state that product must handle any number of records between 1 and 16,383 (2^{14}–1)
  - If system can handle 34 records and 14,870 records, then probably will work fine for 8,252 records
- If system works for any one test case in range (1..16,383), then it will probably work for any other test case in range
  - Range (1..16,383) constitutes an *equivalence class*
- Any one member is as good a test case as any other member of the class

Equivalence Testing (contd)

- 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 fault

Database Example

TEST CASE	NO. OF RECORDS	DESCRIPTION
Test Case 1	0	Member of equivalence class 1 (and adjacent to boundary value)
Test Case 2	1	Boundary value
Test Case 3	2	Adjacent to boundary value
Test Case 4	723	Member of equivalence class 2
Test Case 5	16,382	Adjacent to boundary value
Test Case 6	16,383	Boundary value
Test Case 7	16,384	Member of equivalence class 3

Boundary Value Analysis of Output Specs

- Example:

  In 2001, the minimum Social Security (OASDI) deduction from any one paycheck was $0.00, and the maximum was $4,984.80

  - Test cases must include input data which should result in deductions of exactly $0.00 and exactly $4,984.80
  - Also, test data that might result in deductions of less than $0.00 or more than $4,984.80

Overall Strategy

- Equivalence classes together with boundary value analysis to test both input specifications and output specifications
  - Small set of test data with potential of uncovering large number of faults

Code Walkthroughs and Inspections

- Rapid and thorough fault detection
  - Up to 95% reduction in maintenance costs [Crossman, 1982]

Comparison: Module Testing Techniques

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

Comparison: Module Testing Techniques (contd)

- Tests of 32 professional programmers, 42 advanced students in two groups (Basili and Selby, 1987)
- Professional programmers
  - Code reading detected more faults
  - Code reading had a faster fault detection rate
- 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

Comparison: Module Testing Techniques (contd)

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

Testing Objects

- We must inspect classes, objects
- We can run test cases on objects
- Classical module
  - About 50 executable statements
  - Give input arguments, check output arguments
- Object
  - About 30 methods, some with 2, 3 statements
  - Do not return value to caller—change state
  - It may not be possible to check state—information hiding
  - Method determine balance—need to know accountBalance before, after

Testing Objects (contd)

- Need additional methods to return values of all state variables
  - Part of test plan
  - Conditional compilation
- Inherited method may still have to be tested

Testing Objects (contd)

- Java implementation of tree hierarchy

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

class BinaryTree extends RootedTree
{
  ...
  void displayNodeContents (Node a);
//
// method displayNodeContents defined in this class uses
// method printRoutine inherited from class RootedTree
//
  ...
}

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

Testing Objects (contd)

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

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

- Top half
- When displayNodeContents is invoked in BinaryTree, it uses RootedTree.printRoutine

Testing Objects (contd)

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

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

- Bottom half
- When displayNodeContents is invoked in method BalancedBinaryTree, it uses BalancedBinaryTree.printRoutine

Testing Objects (contd)

- Bad news
  - BinaryTree.displayNodeContents must be retested from scratch when reused in method BalancedBinaryTree
  - Invokes totally new printRoutine
- Worse news
  - For theoretical reasons, we need to test using totally different test cases

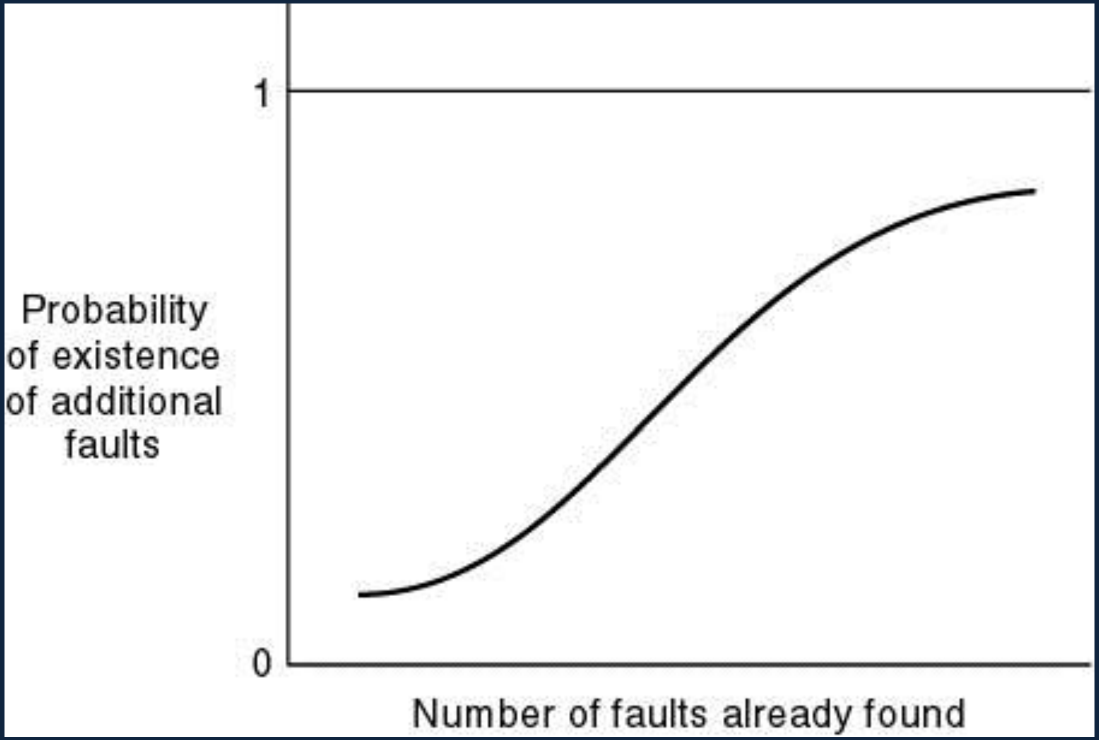
Testing Objects (contd)

- Two testing problems:
- Making state variables visible
  - Minor issue
- Retesting before reuse
  - Arises only when methods interact
  - We can determine when this retesting is needed [Harrold, McGregor, and Fitzpatrick, 1992]
- Not reasons to abandon the paradigm

Module Testing: Management Implications

- We need to know when to stop testing
  - Cost–benefit analysis
  - Risk analysis
  - Statistical techniques

When to Rewrite Rather Than Debug



- When a module has too many faults
  - It is cheaper to redesign, recode
- Risk, cost of further faults

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, recoding

Fault Distribution In Modules Not Uniform (contd)

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

Challenges of the Implementation Phase

- Module reuse needs to be built into the product from the very beginning
- Reuse must be a client requirement
- Software project management plan must incorporate reuse