Software Engineering

CSC440/640
Prof. Schweitzer
Week 7

Requirements Documents Feedback

- Every team should have received feedback on their Requirements Documents
- Make appropriate modifications as part of the final project deliverable
- As you continue in design, if you realize that there are things missing in the requirements, don't be afraid to go back and add them in. That is part of the process.

Choice of Programming Language

- 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?
- 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

- 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

- 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

- 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)?

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

- 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
- 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

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
- 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

- 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

Good Programming Practice

- Additional Reading:
  - Code Complete by Steve McConnell
- Use of consistent and meaningful variable names
  - "Meaningful" to future maintenance programmers
  - "Consistent" to aid future maintenance programmers
  - Example
    - 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

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

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
- What about XML Comments / Javadoc?

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

Code Layout for Increased Readability

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

Nested `if` Statements

```
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";
```

Nested `if` Statements

- 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>
```

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

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
  - FxCop
- Code Reviews End Up Being About Whether You Followed the Standards Instead of Whether You Wrote Good Code
- 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

Drivers and Stubs

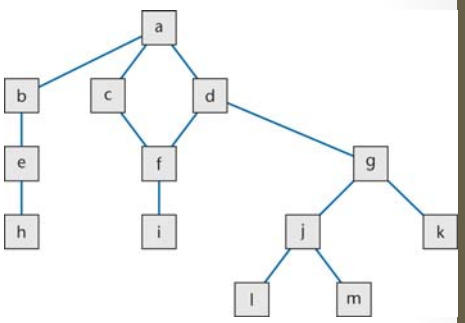
- To test artifact `a`, artifacts `b`, `c`, `d` must be stubs
  - An empty artifact, or
  - Prints a message ("`Procedure radarCalc called`"), or
  - Returns precooked values from preplanned test cases
- To test artifact `h` on its own requires a driver, which calls it
  - Once, or
  - Several times, or
  - Many times, each time checking the value returned
- Testing artifact `d` requires a driver and two stubs

Implementation, Then Integration

- 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
- Solution to both problems
  - Combine unit and integration testing

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`



Top-down Integration

- 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

- 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

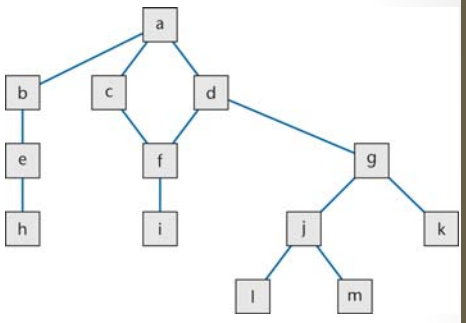
- Problem 1
  - Reusable artifacts are not properly tested
  - Lower level (operational) artifacts are not tested frequently
  - The situation is aggravated if the product is well designed
- Defensive programming (fault shielding)
  - Example:

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

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

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`

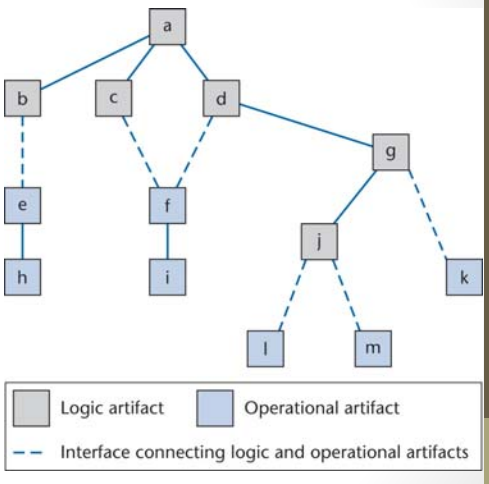


Bottom-up Integration

- 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
- 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

Sandwich Integration

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



Sandwich Integration

- 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

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
- 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

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

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

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 Technique

- 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)

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

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

- 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
- 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

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

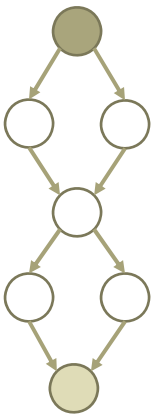
Other Measures of Complexity

- 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
- Useful for Pinpointing Code Needing Review

Cyclomatic Complexity

- Control Flow Structure As a Relative Measure of Complexity
- M = E – N + 2P
  - *M*: Cyclomatic Complexity
  - *E*: Number of Edges
  - *N*: Number of Nodes
    - Conditionals *or* Loops
    - Items that create separate paths through the code
  - *P*: Number of Control Paths
    - Exit Nodes
    - Return Statements
- Tools exist to automatically calculate Cycolmatic Complexity
- Higher Cyclomatic Complexity Requires More Test Cases for Complete Coverage
- High Cyclomatic Complexity Tends to Result in Higher Number of Defects

Cyclomatic Complexity Example



```
public static void Test()
{
   if ( condition1() )
      function1();
   else
      function2();

   if ( condition2() )
      function3();
   else
      function4();
}
```

- Nodes = 7
- Edges = 8
- P = 1
- M = 8 – 7 + (2 x 1) = 3

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

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

Common Version Control Systems

- Locking
  - Source Safe
  - PVCS
- Concurrent
  - CVS
  - Subversion (SVN)
- Distributed
  - BitBucket
  - Git
  - Mercurial
  - Microsoft Team Foundation Server
- There are others… *Many Others!*

Versioning More Than Source

- What about Versioning Documentation?
  - SharePoint
  - Confluence (Jira Studio)
  - Wiki
- Producing File Comparisons Is Difficult
  - Track Changes in Word
  - Revision Page at Front of Document
  - Version Number in Footer of Document

Continuous Integration

- Performs Repeatable Tasks Each Time Code is Committed
- Typical Tasks
  - Build According to Build Scripts
  - Run Unit Tests
  - Code Coverage Analysis
  - Package Executables
  - Label in Version Control
  - Deployment to QA Environment
- Releases Should Only Come from a Build Server
  - Never a Developer's Machine!
- Examples
  - Team City, Cruise Control, Jenkins/Hudson

Next Week

- For Project – Due Next Wednesday (11/2) at 6 PM
  - Class Diagram
  - One Representative Sequence Diagram
- Reading
  - Chapters 5 & 16

Open Forum on Design