Implementation Chapter 15

SEIS 610

Agenda

- Choice of programming language (15.1)
- Good programming practice (15.3)
- Coding Standards (15.4)
- Implementation and Integration (15.6)
  - Top-down, bottom-up, and sandwich implementation and integration (15.6.1, 15.6.2)
- Testing during implementation and Integration (15.9, 15.10, 15.11, 15.3)
- Integration testing (15.20)
- Product testing (15.21)
- Acceptance Testing (15.22)

Choice of Programming Language

- Two times this issue comes up
  - Beginning of a brand new project
  - Rewriting 'ancient' works
    - So essentially a new project
- You would not normally switch in the middle of development
  - Though you might pick a new toolset for the same language

Language 'Taxonomy'

- First generation languages
  - Machine languages
- Second generation languages
  - Assemblers
- Third generation languages
  - High-level languages (COBOL, FORTRAN, C++, Java)
- Forth generation
  - Goal: Anybody can program!

Choice of Programming Language

- Do you really get to pick?
- Normally few choices
  - How is the target programmed?
    - Example: Android uses java
- What development tools are presently used in your organization?

Choice of Programming Language

- Does it ever make sense to move from something the teams are used to?
  - Times change
  - Technologies change
  - Standards change
- Sometime you have to “bight the bullet”

Choice of Programming Language

- Suppose you were moving your application to the 'cloud'
- You could possibly be starting from scratch
- Google App Engine (Platform as a service)
  - Python
  - Java
  - Go

Choice of Programming Language

- If you are switching environments
  - New environment means there **must be some benefit**
  - Does is solve a problem?
  - Does it really make life better?
  - Can it expand your product features or reach?
- There has to be a reason to switch

Choice of Programming Language

- Bottom Line
  - Tools already in house may be adequate, so no choice
  - Target has specific tool requirements, so no choice
  - Long standing problem exists, new language solves
    - Example, designing UI's with Cobol really hard
    - Designing UI's with C# really easy

Next slides form

- http://blog.teamtreehouse.com/choose-programming-language
- Discussion of languages

HTML/CSS

- People often begin by learning HTML and CSS. Why? These two languages are essential for creating static web pages. HTML (Hypertext Markup Language) structures all the text, links, and other content you see on a website.
- CSS is the language that makes a web page look the way it does—color, layout, and other the visuals we call style.
- Really no “logic artifacts”

PHP

- PHP is one of the most popular web languages.
- It runs massive sites such as Facebook and Etsy. WordPress and Drupal are both written in PHP, and those two platforms power a huge number of the sites online today.
- Because of its popularity, learning PHP will serve you well if you intend to code for the Web.

Java Script

- JavaScript is the first full programming language for many people.
- Why? It is the logical next step after learning HTML and CSS. JavaScript provides the behavior portion of a website. For example, when you see a form field indicate an error, that's probably JavaScript at work.
- JavaScript has become increasingly popular, and it now lives outside web browsers as well.
- Learning JavaScript will put you in a good place as it becomes a more general-purpose language.
- JavaScript seems to be everywhere lately.

Java Script

- An important component of cloud computing
- HTML5
- Etc.

Python

- Python is a general-purpose language
  - used for everything from server automation to data science.
- Python is a great language for beginners
  - it is easy to read and understand.
- You can also do so many things with Python that it's easy to stick with the language for quite a while before needing something else.
- Python finds itself at home both creating Web apps like Instagram and helping researchers make sense of their data.

Python

- Google App Engine likes Python!

Ruby

- Ruby is often associated with the Rails framework that helped popularize it.
- Used widely among web startups and big companies alike, Ruby and Rails jobs are pretty easy to come by.
- Ruby and Rails make it easy to transform an idea into a working application, and they have been used to bring us Twitter, GitHub, and Treehouse.

Swift

- Apple released Swift in June, 2014 as a modern language for developing Mac, iPad, iPhone, Apple Watch, and Apple TV applications.
- If you want to enter the world of iOS, Swift is the language with which Apple intends to move forward.
- Yes, many apps are already written in Objective-C, but Swift is here to stay.
- If the Apple ecosystem lures you in, you'll need some understanding of both Objective-C and Swift.

Objective-C

- Like Java, Objective-C can be used to write desktop software and mobile apps. However, Objective-C is essentially Apple territory.
- Until the recent release of the Swift programming language, Objective-C was the language for developing native iPhone and iPad apps.
- Many major apps are still written in Objective-C, and programmers for these apps are in high-demand.
- If you want to work on iPhone and iPad apps, it's a good idea to learn Objective-C.

Java

- Despite its name, Java is not related to JavaScript in any meaningful way.
- Java can be used for anything from web applications to desktop and mobile apps.
- Java has a strong presence among large enterprise applications—think bank, hospital, and university software.
- It *still* powers Android apps, so it's a good choice for those inclined toward mobile development.

Kotlin

- Addresses several Java issues
- Does not mimic java / c syntax
- Secondary (as of now) language for Android

```
fun eatACake() = println("Eat a Cake")
fun bakeACake() = println("Bake a Cake")

fun main(args: Array<String>) {
    var cakesEaten = 0
    var cakesBaked = 0

    while (cakesEaten < 5) {                    // 1
        eatACake()
        cakesEaten ++
    }

    do {                                        // 2
        bakeACake()
        cakesBaked++
    } while (cakesBaked < cakesEaten)

}
```

https://play.kotlinlang.org/byExample/02_control_flow/02_Loops

Good Programming Practice (15.3)

- Use of consistent and meaningful variable names
- “Self documenting code”
  - We always thought this a joke.
- Avoid “magic numbers” (15.3.3)
  - “Use of parameters”
- Code layout (15.3.4)
- Nested if (15.3.5)

Good Programming Practice (15.3)

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

Good Programming Practice (15.3)

- 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

Good Programming Practice (15.3)

- 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
- So be consistent!

Good Programming Practice (15.3)

- **"Self documenting code" --- We always thought this a joke.**
- **You should never need to write a comment to explain what you're doing.**
- **If you feel you have to, rewrite the code until it doesn't need explaining in a comment.**

```
# this method manipulates dimensions
def method_x(a, b, c, d)
  # a is the width
  # b is the height
  # c is the depth
  # d is a list of options
  ...
end
```

That could obviously be re-written as:

```
def manipulate_dimensions(width, height, depth, options)
  ...
end
```

https://news.ycombinator.com/item?id=4381371

Good Programming Practice (15.3)

- 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

Good Programming Practice (15.3)

- Avoid “magic numbers” (15.3.3) -“Use of parameters”
- A 'magic number' is a literal value that appears in a program. For example:

```
total = 1.08 * price;
```

- 1.08 is a magic number because it appears out of the blue, and it's unclear from this line of code what it means. It's generally better to replace magic numbers with NamedConstants, e.g.

```
const double TAX_RATE_IN_TEXAS = 1.08;
total = TAX_RATE_IN_TEXAS * price;
```

- Even better, this should be stored in a configuration file
- http://c2.com/cgi/wiki?MagicNumber

Good Programming Practice (15.3)

- Code layout (15.3.4)
- Each language has an established layout
- Easy to find guidelines
- http://tech.dolhub.com/article/computer/code-Layout
- http://geosoft.no/development/javastyle.html
- http://www.infoq.com/news/2013/08/objective-c-coding-style

Code Layout – nice to read??

```
        try (BufferedReader inputReader = Files.newBufferedReader(
                        Paths.get(new URI
                          ("file:///C:/home/docs/users.txt")),
                        Charset.defaultCharset());
                    BufferedWriter outputWriter = Files.
newBufferedWriter(
                        Paths.get(new URI("file:///C:/home/docs/
users.bak")),
                        Charset.defaultCharset())) {
             String inputLine;
             while ((inputLine = inputReader.readLine()) != null) {
                 outputWriter.write(inputLine);
                 outputWriter.newLine();
}
             System.out.println("Copy complete!");
}
catch (URISyntaxException | IOException ex) {
             ex.printStackTrace();
}
```

From google images

Nested if

```
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";
```

- So how do you test something like this?
- How do you visually verify it is correct?

```
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";
```

Good Programming Practice (15.3)

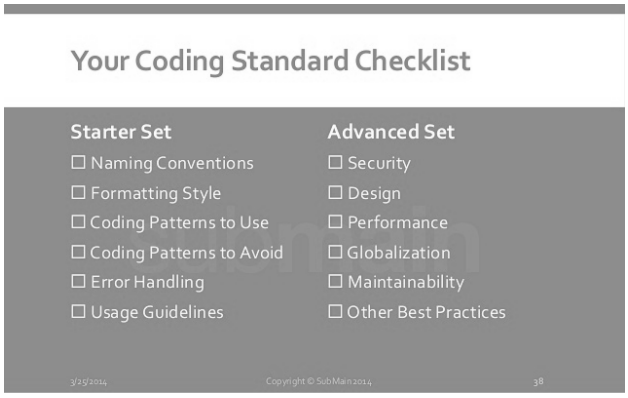
- Nested if (15.3.5)

```
public Boolean userHasAccessToFeature( User u, Feature f) {

        Boolean result = false;
        if ( u.isAdminUser() || u.isSystemAccount()) {
                result = true;
        }
        else {
                Boolean isSecured = f.isSecured();
                if ( !isSecured) {
                        result = true;
                }
                else {
                        List<Integer> acceptableProfiles = f.getAssignedProfiles();
                        If ( acceptableProfiles != null && acceptableProfiles.size() > 0) {
                                List<Integer> assignedProfiles = u.getAssignedProfiles();
                                If ( assignedProfiles != null && assignedProfiles.size() > 0) {
                                        for (int idx = 0; idx < assignedProfiles.size(); idx++) {
                                                Integer profile = assignedProfiles.get(idx);
                                                if ( acceptableProfiles.contains( profile) {
                                                        result = true;
                                                        break;
                                                }
                                        }
                                }
                        }
                }
        }
}
```

Coding Standards (15.4)

- Bundles all the above into a nice neat package
- Your organization might have one
- Ask for it.



Remarks on Programming Standards

- The aim of standards is to make maintenance easier
  - If they make development difficult, then they must be modified
  - Overly restrictive standards are counterproductive
  - The quality of software suffers

15.6.1 Top-down Integration

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

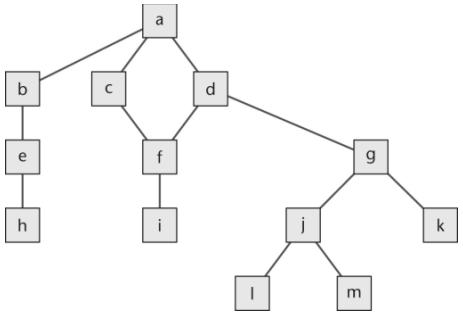


Figure 15.6 (again)

Top-down Integration (contd)

- 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 (necessarily) 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);
```

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

15.6.2 Bottom-up Integration

- If code artifact `mAbove` calls code artifact `mBelow`, then `mBelow` is implemented and integrated before `mAbove`
- One possible bottom-up ordering is

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

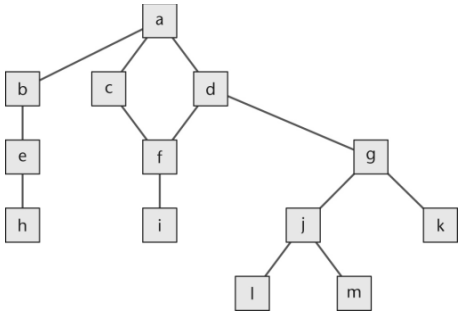


Figure 15.6 (again)

15.6.2 Bottom-up Integration

- Another possible bottom-up ordering is

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

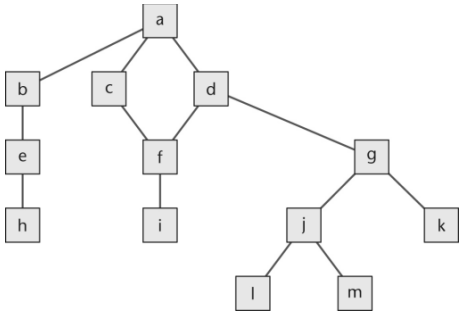


Figure 15.6 (again)

Bottom-up Integration (contd)

- Advantage 1
  - Operational artifacts are thoroughly tested
- Advantage 2
  - Operational artifacts are tested with drivers, not by fault shielding, defensively programmed artifacts
- Advantage 3
  - Fault isolation

Bottom-up Integration

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

15.9 The Test Workflow: Implementation

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

15.10 Test Case Selection

- Worst way — random testing
  - "Haphazard test data"
  - There is likely no time to test all paths
  - **BUT random data testing not completely bad**
- We need a systematic way to construct test cases

15.10.1 Testing to Specifications versus Testing to Code

- There are two extremes to testing
- *Test to specifications* (also called black-box, data-driven, functional, or input/output driven testing)
  - Ignore the code — use the specifications to define test cases
- *Test to code* (also called glass-box, logic-driven, structured, or path-oriented testing)
  - Ignore the specifications — use the code to define test cases

15.10.2 Feasibility of Testing to Specifications

- Example:
  - The specifications for a data processing product include 5 types of commission and 7 types of discount
  - 35 test cases
- We cannot say that commission and discount are computed in two entirely separate artifacts
- Remember: blackbox testing… the structure is irrelevant

Feasibility of Testing to Specifications

- Suppose the specifications include 20 factors, each taking on 4 values
  - There are 4^{20} or 1.1 *10^{12} possible 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 impractical

15.10.3 Feasibility of Testing to Code

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

Feasibility of Testing to Code

- Code example:

```
read (kmax)                          // kmax is an integer between 1 and 18
for (k = 0; k < kmax; k++) do
{
   read (myChar)                     // myChar is the character A, B, or C
   switch (myChar)
   {
     case 'A':
       blockA;
       if (cond1) blockC;
       break;
     case 'B':
       blockB;
       if (cond2) blockC;
       break;
     case 'C':
       blockC;
       break;
   }
   blockD;
}
```

Figure 15.9

Feasibility of Testing to Code

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

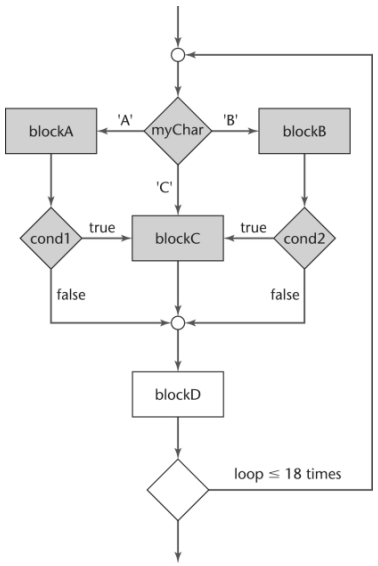


Figure 15.10

Feasibility of Testing to Code

- Testing to code may not be 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

- We can exercise every path without detecting every fault

Feasibility of Testing to Code

- A path can be tested only if it is present
- A programmer who omits the test for d = 0 in the code probably is unaware of the possible danger

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

(a)

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

(b)

Figure 15.12

Feasibility of Testing to Code

- Criterion "exercise all paths" is not *reliable*
  - Products exist for which some data exercising a given path detect a fault, and other data exercising the same path do not

15.11 **Black-Box** and 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 and Unit-testing Techniques

- 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

- Any one member of an equivalence class is as good a test case as any other member of the equivalence class
- Range (1..16,383) defines three different equivalence classes:
  - Equivalence Class 1: Fewer than 1 record
  - Equivalence Class 2: Between 1 and 16,383 records
  - Equivalence Class 3: More than 16,383 records

Boundary Value Analysis

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

Equivalence Testing of Output Specifications

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

  In 2008, the minimum Social Security (OASDI) deduction from any one paycheck was $0, and the maximum was $6,324
  - Test cases must include input data that should result in deductions of exactly $0 and exactly $6,324
  - Also, test data that might result in deductions of less than $0 or more than $6,324
  - **Test cases for something in between!**

Overall Strategy

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

15.11.2 Functional Testing

- Functions are tested by feeding them input and examining the output, and internal program structure is rarely considered.

15.11.2 Functional Testing

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

Functional Testing

- 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

15.13 Glass-Box *Unit-Testing* Techniques

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

15.13.1 Structural Testing: Statement, Branch, and Path Coverage

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

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

Test case: s = 2, t = 0.

Figure 15.15

Structural Testing: Branch Coverage

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

Structural Testing: Path Coverage

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

Linear Code Sequences

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

All-Definition-Use-Path Coverage

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

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

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

All-Definition-Use-Path Coverage

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

Infeasible Code

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

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

(a)

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

(b)

15.13.2 Complexity Metrics

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

Lines of Code

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

Other Measures of Complexity

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

Problem with Complexity Metrics

- Complexity metrics, as especially cyclomatic complexity, have been strongly challenged
- SO WHAT
- It is a tool and can be a useful tool.
- It should not make final decisions for you

Code Walkthroughs and Inspections

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

15.20 Integration Testing

- The testing of each new code artifact when it is added.
  - to what has already been tested
- Special considerations testing graphical user interfaces

Integration Testing of Graphical User Interfaces

- GUI test cases include
  - Mouse clicks, and
  - Key presses
- These types of test cases cannot be stored in the usual way
  - We need special tools
- Examples:
  - Selenium
  - http://www.seleniumhq.org/

15.21 Product Testing

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

Product Testing for Custom Software

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

Product Testing for Custom Software

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

15. 22 Acceptance Testing

- The client determines whether the product satisfies its specifications
- Not that just if the product is broken, but does it do what the client needs

- Acceptance testing is performed by
  - The client organization, or
  - The SQA team in the presence of client representatives, or
  - An independent SQA team hired by the client

Acceptance Testing (contd)

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

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

Acceptance Testing (contd)

- The key difference between product testing and acceptance testing is
  - Acceptance testing is performed on actual data (maybe)
  - Product testing is preformed on test data, which can never be real, by definition
  - Acceptance testing is performed to see that the software does what it is supposed to, not just that it works.

The End