Session 8: Testing
Special importance of testing
Stages of testing
Manual vs. automated testing
Continuous testing strategies

Why is testing critical to project management?
- It's when many projects first fall behind schedule and go over budget
  - Everything looks good until we get into heavy late-stage testing, which reveals unexpected bugs.
  - Results may require substantial rework, thus adding many unplanned tasks to the plan!
  - In extreme case we may lose control of the project!
- An I.T. project manager really needs to understand all this.
- Why doesn't our SDLC have a testing phase?

Side note
- Before testing, a dishonest or incompetent project manager can often bluff the clients and upper management about status.
- But test results are visible. Anyone can see whether they're valid. Bluffing is over!

What's wrong with this?
- The programming is done. Now we're ready to begin testing.
- An old-fashioned naive view. Is programming really done?
**Programming**

- Programming = internal component design
  - + coding
  - + testing

- A single task on the project plan
  
  *Develop* (or program) **glop33** means
  
  1. Design **glop33**
  2. Code & unit test **glop33**
  3. Deliver thoroughly tested **glop33** for integration into the application system

  *What about documenting?*

**Two kinds of documentation for a component or program**

- **Usage** (external) documentation:
  - Everything a user needs to know in order to make proper use of the program.

- **Maintenance** (internal) documentation
  - Everything a programmer needs in order to
    - understand how it works
    - diagnose possible malfunctions
    - make corrections and enhancements

  *Which is more important? Which should each be prepared?*

**Stages of software testing**

- For *all* major software development:
  - **Unit** test
  - **Integration** test
  - **System** test
  - **Volume** (stress) test
  - **Acceptance** test and start-up

- For a software *product* also (optional)
  - **Alpha** test
  - **Beta** test

**Unit testing**

- Validates a **single component**
  - (MUT: module under test)
    - a non-trivial subroutine or function
    - an object-oriented class
    - other (anything worth assigning and keeping track of)

- It is inseparable from coding
  - It’s (almost) always done by the same programmer
  - as part of a single task "Develop xyz"

- Automated vs. manual unit testing
**Manual unit testing**

- In a manual test, the programmer
  - launches execution
  - enters test-case data
  - examines results
    - if unexpected, make corrections and redo
- How do we assure adequate test coverage?
- How do we know a future change won't invalidate a previous test result?

**Bottom-up unit testing**

- The programmer writes a special **test driver** program to invoke the MUT.
  - Testing can be extremely thorough, since the driver can invoke the MUT with a wider range of inputs than the eventual application will encounter.
  - There's no place in the eventual application program for the driver, so retesting the MUT may require special set up. In any case **don't throw the driver away when you finish the test**.
- The MUT may invoke other modules, which presumably have already undergone thorough unit test.

**Automated unit testing**

- The programmer
  - writes code to invoke MUT with specified inputs
  - writes code to receive the result and compare it with the expected value
  - launches execution which will:
    - Do nothing (or just report OK) if actual result matched expected result.
    - Issue alarm if not
  - reruns the same tests **every time anything is changed**.
- A related fad is TDD (**test-driven development**) **How does that work?**

**Automated unit-testing frameworks**

- **JUnit** for Java and similar products for other languages reduce the amount of code you have to write for an automated unit test.
- But they don't guarantee thoroughness, and they may generate huge test drivers.
- Use them when they suit your needs.
Which comes first?
- The MUT or the test drivers?
- Test-driven development (TDD) recommends that you code automated test drivers before coding (or even designing) the MUT.
  - Advantages:
    - Focuses the programmer's attention on what the MUT must produce.
    - Facilitates incremental development.
  - Disadvantages:
    - May have to be redone when the MUT is changed, esp. its interface.
    - May distract from thorough algorithm analysis.

Integration testing
- Validates the interfaces among the components in the SUT (system under test)
- May be continuous or ad hoc

Top-Down testing
- Combines unit testing with integration testing.
- The MUT is invoked by the actual real higher-level module in the application, which has already been tested.

How is that possible?

Top-Down testing (continued)
- The programmer codes dummy modules ("stubs") to simulate lower-level modules that the MUT needs to invoke.
  - They can be thrown away later.
  - They yield a constant or trivial result, often wrong but always legal.

Why?
- Testing of the interfaces
  - is more certain than with bottom-up unit testing,
  - but range of test cases will usually be narrower.
Continuous integration testing

- Not really *continuous* but *recurring* on a regular schedule (usually overnight)
- A complete "build" of the current application:
  - Dozens, maybe hundreds of compilations
  - May take several hours and use huge resources
  - Experiments with parallel tasks
  - Driven by Make file or equivalent control
- A single faulty module can invalidate a test!
  - How can we keep that from happening often?
  - How should we react when it happens?

Advantages & disadvantages?

Automated integration testing

- Just as we could automate unit tests by building test cases into the driver program, we can automate integration test
  - There are software products that simulate user transaction entry.
- Following each complete system *build*, we rerun the tests and get a failure report.
- Some organizations do this every night! (That generates a lot of pressure on the programming staff.)

Volume (or stress) test

- Tests the limits of
  - transaction volume
  - number of simultaneous users
  - database size
  - bizarre combinations of data entry, including the most diabolical user data-entry errors anyone can think of.
- Tests *performance* as well as correctness
  - Transaction response times
  - Start-up & initialization and shut down
- This is the time for *professional testers*
  - sometimes (misleadingly) called "Q.A."

What next?

- If the project's end product is to be an *application system* for a sponsoring end user, proceed to *system test* and *acceptance test*.
- If the project's end product is to be a *software product* that will be marketed outside the company, consider *alpha test* and *beta test*. 
System testing

- For the first time tests all components of the new application system together

- Includes the **people** and **user documentation** components
  - **Actual** user personnel
  - No I.T. hand-holding or coaching

- Includes all processing cycles, such as
  - end-of-month, end-of-year accounting close
  - disaster recovery procedures

Note the progression of participation in testing

<table>
<thead>
<tr>
<th>Test stage</th>
<th>Run mainly by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integration</td>
<td>Programmers</td>
</tr>
<tr>
<td>Stress</td>
<td>Professional testers</td>
</tr>
<tr>
<td>System</td>
<td>User personnel</td>
</tr>
</tbody>
</table>

Preparing for the system test and the acceptance test

- The project plan must contain detailed tasks to assure that
  - Participating user personnel get appropriate training just in time.
  - All participants know exactly what they're going to do (script).

What does that mean?

- Never **improvise** during a system test!

    Why not?

Preparing the user organization

- The **systems analyst** can play the major role in preparing the user organization
  - That gives the analyst something important to do after defining the requirements (ESD)
  - The analyst usually has the necessary knowledge to take charge of this area.

- Do users always hope for success?
Acceptance testing

- May occur as the final portion of system test
- Does the system do what the specifications (ESD) call for?
  - If so, we're ready for installation and start-up
  - If not, we have a list of urgent corrections to be made, and (at least) the acceptance testing must be repeated.
    - Important: Resist temptation to take shortcuts here!

Result of acceptance test

a. No problems exposed; users are 100% satisfied with the application system.
   - Proceed to installation & start-up

b. Trivial bugs exposed; user-management willing to "live with" them until corrected through routine maintenance.
   - Proceed to installation & start-up

c. Serious bugs or performance problems exposed.
   - Reschedule installation to occur after thorough re-test

Proceeding to installation

- Installing a new application system usually involves running some one-time programs for:
  - Converting data from old system files to the new system's files or database
  - Initializing certain data.

- Of course, those one-time programs must themselves have been thoroughly tested.
  - Surprisingly, many serious failures have occurred because of unexpected conversion problems encountered during installation!

Alpha and Beta testing

- When a software product is going to be released to the market we can gain more confidence by first:
  - releasing it to internal users within our company (alpha test). Once that goes well, then
  - releasing it to friendly volunteer outside users who agree to report problems but promise not to complain or sue or write reviews (beta test).

- No matter how thorough we thought the system and volume tests were, actual use by real people usually turns up problems