**Week 1: What is software quality?**

- Aspects of software quality
- Stages of software testing

---

**Two aspects of software quality**

<table>
<thead>
<tr>
<th></th>
<th>External Quality</th>
<th>Internal Quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Measured by:</td>
<td>Operational use</td>
<td>Maintainability</td>
</tr>
<tr>
<td></td>
<td>- reliability/correctness/absence of defects</td>
<td>- ease of understanding</td>
</tr>
<tr>
<td></td>
<td>- ease/naturalness of use</td>
<td>- ease / cost of making changes</td>
</tr>
<tr>
<td>Perceived mainly by:</td>
<td>End users</td>
<td>Maintenance programmers</td>
</tr>
</tbody>
</table>

- Which does our textbook emphasize?
- Which is more important? Why?
- Are they mutually independent?

---

**Evolving quality over the lifespan of a program**

- **External** quality usually gets **better** over the lifespan of a program.
  - Serious bugs eventually get fixed

- But **internal** quality often gets **worse** as corrections and enhancements are integrated into poorly organized code.
  - Common symptoms:
    - misplaced functions
    - unnecessary duplication of code
    - failure to handle special cases

---

**A 1960's programming manager's cliché**

*Any program that works is better than any program that doesn’t work!*

 Agree or disagree?
A popular saying (agree or disagree?)

- "Testing can only show the presence of defects, not their absence."
- Questions:
  - Does that mean we can never say that a program is "fully tested"?
  - What is the main influence on the thoroughness of software testing?
- Those with a positive outlook on testing prefer the term "software validation", esp. over debugging.

Inexcusable program bugs

- A few surprisingly common bugs are so obvious and so easy to avoid that we can consider them inexcusable (even a firing offense)! They include:
  - Buffer overflow
  - Misinterpreted unit of numeric measure
  - Calendar miscalculations, e.g. leap year
  - Mismatched text box capacity and data-item size
  - Failure to validate data item from an external source (keyboard, Internet, etc.)

  - Is that fair?

Two strategies for software validation

A. Prepare and run extremely thorough and exhaustive test cases.

B. Follow programming practices that minimize the likelihood of errors in the first place.

Such as?

Which should we emphasize?

The structured revolution ca. 1975

- Collection of concepts & techniques aimed at enhancing program reliability and maintainability:
  - Highly-modular program structure (enhanced later by OOP)
  - Structured (go-to-less) code
  - Top-down design
  - Data-representation standards
  - High degree of component reuse
  - Table-driven logic
  - Inspections & reviews
  - Egoless team organization
  - etc.

- It was then claimed that, with competent programmers, the results could be close to error-free!

  How did that turn out?
Two strategies for software validation

A. Prepare and run extremely thorough and exhaustive test cases.
B. Follow programming practices that minimize the likelihood of errors in the first place.

We have to emphasize both!

Note that the best possible A can't salvage a bad B

Acceptable levels of software reliability

How different would the thoroughness of verification be:
- for an airline's employee newsletter application?
- for an airline's payroll system?
- for an airline reservation system?
- for an aircraft maintenance record application?
- for an air-traffic-control system?

Stages of software testing
(after specification and general design are solid)
- Unit test
- Integration test
- System test
- Alpha test
- Beta test
- Stress test
- Acceptance test

When does each occur?
What is being validated?
Who has main responsibility?
Who else participates?

Does this sequence of 3 tasks in a project plan make good sense?

If so, why?
If not, why not?
These three activities are highly integrated and usually inseparable

- They're almost always assigned to the same individual project team member.
- Results of one may require redoing earlier ones.
- Here's how it usually appears on a sensible project plan:

![Diagram showing the sequence of activities: Detailed design module Q → Code module Q → Unit test module Q → Develop module Q]

Two strategies for unit & integration testing of a well-designed (highly modular) application

- Bottom-up development
- Top-down development

Both require highly modular program structure.

Bottom-up module development

- Starting with lowest level modules, i.e., that don't need or use any undeveloped modules:
  - Develop a driver program to exercise the MUT
  - Is that a temporary "throw-away" program?
  - After the MUT is fully tested, move up to higher-level modules that may use it.
- This is a traditional approach for developing modular programs.
  - What other kind of program is there?

Top-down module development

- Starting with the highest level (i.e., modules that aren't called by other modules):
  - Develop a dummy stub module for each module that it calls, returning a trivial or constant result.
  - Are those temporary "throw-away" modules?
  - After the high-level module works satisfactorily, start replacing the stub modules with the real ones, applying the same technique.
- A somewhat newer technique than bottom-up.
  - Is it always possible?
Advantages

- **Bottom-up development**
  - Test can be more thorough; the driver can exercise more cases than the final real high-level module.
  - Fits better with object-oriented approach (why?)
  - Facilitates module re-use (why?)

- **Top-down development**
  - Unit test and integration test are tightly integrated; we test the internals and the interfaces together.
  - Evolves from user prototyping.
  - Often takes less time.
  - May expose faulty design sooner.

Non-modular ("monolithic") software design

- This is not a controversial topic among knowledgeable programmers.
  - *No experts* advocate monolithic design today.

- Nevertheless:
  - You will find examples in some older ("legacy") software.
  - Some unenlightened programmers are still developing such code. They may not understand what we mean when we ask about **top-down vs. bottom up** (or even about modular structure).
  - The COBOL language encouraged monolithic organization.

Exercise / demonstration

- Let's examine a couple of small independent C# functions (handout #1, also in Blackboard).
  - **greatest common divisor** of 2 integers \( \text{gcd}(m, n) \)
  - **least common multiple** of 2 integers \( \text{lcm}(m, n) \)
    - Note that \( \text{lcm} \) uses \( \text{gcd} \)

- First let's test it by interaction from the console with command-line parameters.
  - How do we feel now about the functions' reliability?

Assignment #1: background

- Look over the MUT (\( \text{gcd} \) & \( \text{lcm} \)).
  - Why are Java and C# better suited to this problem than C++? (a subtle difference)
  - Algorithm for \( \text{gcd} \) is recursive.
  - Examine an iterative implementation (e.g. 6.8.4 in introcs.cs.luc.edu/html/)
  - Which do you prefer? Why?
  - Is there an obvious flaw (bug)?
Assignment #1: getting started

- Now look over the test driver (main)
  - What are its strengths and weaknesses?
  - Run the program
  - How confident are we that the MUT works perfectly?

Assignment #1

- Develop and run 2 new versions of the test driver program:
  - Console dialog to run a user-controlled number of test cases
  - Fully automated verification--no console interaction.

- If you discover a flaw in the MUT (gcd or lcm), by all means correct it. Otherwise leave them alone.