**Week 4: Test Case Design**

- The tester's goal
- Popular strategies
- Exhaustive testing

"Probably the poorest methodology of all is random-input testing--the process of testing a program by selecting, at random, some subset of all possible input values."

- Glenford Myers, p. 36

Do you agree? Why

What would be better?

---

**The primary goal**

- "Testing is the process of executing a program with the intent of finding errors. . . If our goal is to demonstrate that a program has no errors, then we shall subconsciously be steered toward this goal."

- Myers, p. 5

Do we believe that?

Does that imply that the tester should be someone other than the programmer?

- "A successful test case is one that detects an as-yet undiscovered error."

- Myers, p. 16

---

**Controversial (possibly troublesome) advice from an expert**

- "A programming organization should not test its own programs"

- Myers, p. 13

Why not?

What kind of programs?

What kind of testing?
Two popular strategies

- **Black-box testing**
  - Tester is unconcerned about MUT's internal behavior and structure.
  - Exhaustive testing (all possible inputs) is impossible for non-trivial MUT.
  - See Ammann & Offutt, p. 21

- **White box testing**
  - Tester examines internal structure of MUT and contrives cases to exercise paths.
  - Exhaustive testing is still impossible, but we can often come closer.

*How?*

Which is better? easier? safer? Which should we prefer?

Sequence of activity

- **With black-box testing** one can code the test cases **before** developing the MUT
  - This is called test-driven or test-first development (TDD or TFD)
  - Is there a difference?

- **Arguments for:**
  - Focuses attention on the specifications
  - TFD (not TDD) facilitates dividing responsibility

- **Arguments against:**
  - Encourages hasty (sloppy?) implementation.
  - May lock into inappropriate design (e.g. fig 10.1).
  - Omits "white box" cases

*What else?*

When should we prepare a test suite?

a. When we first identify the need for the MUT, as an aid to designing its interface (TDD)?

b. As soon as the MUT's interface is designed, before any MUT code is written (TFD)?

c. After the MUT is mostly coded and ready to try to run?

*Your opinion?*

Testing error conditionss

- Don't forget to test the MUT's behavior when it encounters erroneous data. Verify that cases that shouldn't work don't yield misleading results.

- Distinguish between an **error** and a **bug**.

- An error can be:
  - **fatal**: abort the run with as much helpful information as possible.
  - **serious**: issue meaningful error diagnostic message and skip this case (e.g. transaction)
  - **minor**: issue warning message

*Which call for throwing an exception?*
Helpful tools

- Different programming languages provide different tools.
  - They've evolved over many years.
  - Some are official parts of the language or standard libraries; others are add-on products.

  Java supports:
  - The `Assert` statement
  - The `junit` run-time framework

C++ supports:
- The `Assert` macro
- Preprocessor facilities

Be familiar with these.

Some C# debugging tools

- Inspired by similar facilities in Java, etc.
- You don't really need them, and this course won't require you to use them.
- But feel free use them if you're comfortable with them
  - Assertion functions: (see 14.1)
    - `Assert.IsFalse(condition)` Terminates task with error message if condition (Boolean expression) is false.
  - NUnit unit-test framework (see 14.8)
    - Don't complain if they don't work.

Object mutability

- An object is said to be **immutable** if its value cannot be changed once it's created (usually by a constructor)
  - Java's `final` attribute and C++'s `const` help to enforce immutability.
- Immutability can simplify testing and enhance operational reliability.
  - It's strongly recommended by J. Bloch and others.
- But as Scott Meyers points out it can also degrade performance, by forcing unnecessary creation of new objects.
  - Let's look at an example

Example

- C++: Suppose `a` and `b` are `Angle` objects
  - `a = a + b;` is much slower than `a += b;`
    - Why?

  - Java: Compare the equivalent
    - `a = a.add(b);`
      - with `a.addSet(b);`

  - In testing we're interested both in correct results and tolerable performance.
Example
- C++: Suppose a and b are Angle objects
  - `a = a + b;` Creates new object `a+b`
    is much slower than
  - `a += b;` Modifies existing object `a`

- Java: Compare the equivalent to the above
  - `a = a.add(b);`
    with
  - `a.addSet(b);`

- In testing we're interested both in correct results and tolerable performance.

What does C# do with that example?
- In C# the programmer defines the simple operators (+, -, *, /, %) for a new class, but the compiler (somehow) generates the corresponding compound assignment operator definitions (+=, -=, *=, /=, %=).
- We hope that it figures out how to avoid creating a new object.

A Java and C# irritation
- Java’s library String objects are immutable (C#, too!)
- Programmers and users complained that it was painfully slow to modify String objects, e.g. by repeated concatenation.
- Java responded with StringBuffer, mutable but notationally inconvenient.
  - C# followed with StringBuilder

Was that well thought out in advance?