CSE 331
Software Design & Implementation

Kevin Zatloukal
Spring 2020
Testing
How many tests is enough?

Correct goal should use **revealing subdomains**:  
- one from the middle of each subdomain  
- examples along the boundaries of each subdomain
How many tests is enough?

Common goal is to achieve high **code coverage**:
- ensure test suite covers (executes) all of the program
- assess quality of test suite with % **coverage**
  - tools to measure this for you

**Assumption** implicit in goal:
- if high coverage, then most mistakes discovered
- **far** from perfect but widely used
- low code coverage is definitely bad
int min(int a, int b) {
    int r = a;
    if (a <= b) {
        r = a;
    }
    return r;
}

• Consider any test with $a \leq b$ (e.g., $\text{min}(1,2)$)  
  – executes every instruction  
  – misses the bug

• Statement coverage is not enough
Code coverage: branch coverage

```c
int quadrant(int x, int y) {
    int ans;
    if (x >= 0)
        ans=1;
    else
        ans=2;
    if (y < 0)
        ans=4;
    return ans;
}
```

- Consider two-test suite: (2,-2) and (-2,2). Misses the bug.
- *Branch coverage* (all tests “go both ways”) is not enough
  - here, *path coverage* is enough (there are 4 paths)
int countPositive(int[] a) {
    int ans = 0;
    for (int x : a) {
        if (x > 0)
            ans = 1; // should be ans += 1;
    }
    return ans;
}

• Consider two-test suite: [0,0] and [1]. Misses the bug.
• Or consider one-test suite: [0,1,0]. Misses the bug.
• Path coverage is enough, but no bound on path-count!
Code coverage: what is enough?

```c
int sumOfThree(int a, int b, int c) {
    return a+b;
}
```

- *Path coverage* is not enough
  - consider test suites where \( c \) is always 0

- Typically a “moot point” since path coverage is unattainable for realistic programs
  - but do not assume a tested path is correct
  - even though it is more likely correct than an untested path

- Another example: buggy \texttt{abs} method from earlier in lecture
Varieties of coverage

Various coverage metrics (there are more):

- Statement coverage
- Branch coverage
- Loop coverage
- Condition/Decision coverage
- Path coverage

Limitations of coverage:

1. 100% coverage is not always a reasonable target
   - may be *high cost* to approach 100%
2. Coverage is *just a heuristic*
   - we really want the revealing subdomains for the errors present
Summary of Heuristics

- Split subdomains on boundaries appearing in the specification
- Split subdomains on boundaries appearing in the implementation
- Test boundaries that commonly lead to errors
- Test special cases like nulls, empty arrays, 0, etc.
- Tests to exercise every branch of the code
  - all paths would be even nicer (but not always possible)
- Test any cases that caused bugs before (to avoid regression)

On the other hand, don't confuse volume with quality of tests
  - look for revealing subdomains
  - want tests in every revealing subdomain not just lots of tests
Testing Tools

• Modern development ecosystems have built-in support for testing

• Your homework introduces you to Junit
  – standard framework for testing in Java

• Continuous integration
  – ensure tests pass **before** code is submitted

• You will see more sophisticated tools in industry
  – libraries for creating mock implementations of other modules
  – automated tools to test on every platform
  – automated tools to find severe bugs (using AI)
  – …
Testing Tips

• Write tests both **before** and **after** you write the code
  – (only clear-box tests need to come afterward)

• Be systematic: think through revealing subdomains & test each one

• Test your tests
  – try putting a bug in to make sure the test catches it

• Test code is different from regular code
  – changeability is less important; **correctness** is more important
  – do not write **any test code** that is not obviously correct
    • otherwise, you need to test that code too!
    • unlike in regular code, it’s **okay** to repeat yourself in tests