CSE 303:
Concepts and Tools for Software Development

Hal Perkins

Winter 2009

Lecture 16— Testing

Where are We

- Some very basic "software-engineering" topics in the midst of tools (take CSE 403 for much more)
  - Today: testing (how, why, some terms)
  - Later: (partial) specification

Testing 1, 2, 3

- Role of testing and its plusses/minuses
- Unit testing or "testing in the small"
- Stubs, or "cutting off the rest of the world" (which might not exist yet)

A little theory

- Motto (Hunt and Thomas): "Test your software or your users will"
- Testing is very limited and difficult:
  - Small number of *inputs*
  - Small number of calling contexts, environments, compilers, ...
  - Small amount of *observable output*
  - Requires *more* things to get right, e.g., test code
- Standard *coverage metrics* (statement, branch, path) are useful but only emphasize how limited it is.

3 coverage metrics

```
int f(int a, int b) {
  int ans = 0;
  if(a)
    ans += a;
  if(b)
    ans += b;
  return ans;
}
```

Statement coverage: `f(1,1)` sufficient

Branch coverage: `f(1,1)` and `f(0,0)` sufficient

Path coverage: `f(0,0)`, `f(1,0)`, `f(0,1)`, `f(1,1)` sufficient

But even the example path-coverage test suite suggests `f` is a correct "or" function for C; it is not.

Colored boxes

"black-box" vs. "white-box"

- black-box: test a unit without looking at its implementation
  - Pros: don't make same mistakes, think in terms of interface, independent validation
  - Basic example: remember to try negative numbers
- white-box: test a unit with looking at its implementation
  - Pros: can be more efficient, can find the implementation's corner cases
  - Basic example: try loop boundaries, "special constants"

Stubs

- Unit testing (a small group of functions) vs. integration testing (combining units) vs. system testing (the "whole thing" whatever that means)
- How to test units ("code under test") when the other code:
  - may not exist
  - may be buggy
  - may be large and slow
- Answer: You provide a "fake implementation" of the other code that "works well enough for the tests".
  - Fake implementation is as small as possible, so the functions are often called "stubs".
- Tools like JUnit et seq. exist to support unit testing — take advantage of them when they make sense

Stubbing techniques

It's an art, not a science. Kinds of techniques that are useful:

- Instead of computing a function, use a small table of pre-encoded answers
- Return wrong answers that won't mess up what you're testing
- Don't do things (e.g., print) that won't be missed
- Use a slower algorithm
- Use an implementation of fixed size (an array instead of a list?)
- ... other ideas?

Lecture-size example can be tough, but we can try to get the idea across.

Eating your vegetables

- Make tests:
  - early
  - easy to run (e.g., a make target with an automatic diff against sample output)
  - that test interesting and well-understood properties
  - that are as well-written and documented as other code
- Write the tests first (seems odd until you do it)
- Write much more code than the "assignment requires you turn-in"
- Manually or automatically compute test-inputs and right-answers?
- Write *regression tests* and run on *each version* to ensure bugs do not creep in for stuff that "used to work".

Testing – of what

Summary: Testing has some concepts worth knowing and *using*

- Coverage (statement, branch, path)
- White-box vs. black-box
- Stubbing

But we made a *big* assumption, that we know what the code is *supposed* to do!

Specification is a topic we need to talk more about . . .

. . . and we will, later.