CSE 303:
Concepts and Tools for Software Development

Dan Grossman

Winter 2006

Lecture 17— Unit testing, stubs, specification, etc.

Where are We

- In the middle of software development tools
  - "Done": preprocessors, compilers, debuggers, profilers,
  - "To do": compilation-managers, version-control systems, linkers, archive-generators
- Today: "software engineering" topics related to homework 5.

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.

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, indepdent 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”.

Stubbing techniques

Honestly something I've never been taught, but here are some tricks I use:

- 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 show the ideas with the prime-number code from last lecture.

Eating your vegetables

- Make tests:
  - early
  - easy to run
  - that test interesting and well-understood properties
  - that are as well-written and documented as other code
- Write the tests first?
- Write much more code than the “assignment requires you turn-in”
- Manually or automatically compute test-inputs and right-answers?

Testing – of what

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

- Coverage
- White-box vs. black-box
- Stubbing

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

Oftentimes, a complete *specification* is as difficult as writing the code. But:

- It's still worth thinking about.
- *Partial* specifications are better than none.
- *Checking* specificatins (at compile-time and/or run-time) is great for finding bugs early and "assigning blame".

Full Specification

Often tractable for very simple stuff: “Take an `int` x and return 0 iff there exists ints y and z such that $y * z == x$ (where $x, y, z > 0$ and $y, z < x$).

What about sorting a doubly-linked list?

- Precondition: Can input be NULL? Can any `prev` and `next` fields be NULL? Must it be a cycle or is “balloon” okay?
- Postcondition: Sorted (how to specify?) – *and* a permutation of the input (no missing or new elements).

And there’s often more than “pre” and “post” – time/space overhead, other effects (such as printing), things that may happen in parallel.

Specs should guide programming and testing!

Partial Specifications

The difficulty of full specs need not mean abandon all hope.

Useful partial specs:

- Can args be NULL?
- Can args alias?
- Are stack pointers allowed? Dangling pointers?
- Are cycles in data structures allowed?
- What is the minimum/maximum length of an array?
- ...

Guides callers, callees, and testers.

Beyond testing

Specs are useful for more than “things to think about while coding” and testing and comments.

Sometimes you can check them dynamically, e.g., with *assertions* (all examples true for C and Java)

- Easy: argument not NULL
- Harder but doable: list not cyclic
- Impossible: Does the caller have other pointers to this object?

Or statically using stronger type systems or other tools:

- Plusses: earlier detection (“coverage” without running program), faster code
- Minus: Potential “false positives” (spec couldn’t ever actually be violated)