CSE 303:
Concepts and Tools for Software Development

Dan Grossman

Spring 2005

Lecture 19— Unit Testing, Stubs, ...

Where are We

- In the middle of software development tools
  - "Done": preprocessors, compilers, debuggers, profilers, linkers, archive-generators
  - "To do": compilation-managers, version-control systems
- Wednesday: The ACM code of ethics
  - Please read/skim and think about justification and disagreement
- Today: A "software engineering" topic very 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.
- “black-box” vs. “white-box”

A comparsion – typechecking

In a “strongly typed” language (e.g., Java) typechecking:

- gets much better “coverage” (any caller in Java with any correct compiler and any arguments)
- of much weaker properties (you get a `Foo` back, unless you return `null`, throw an exception, run out of stack or heap space, enter an infinite loop, ...

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?

Isolating the code

The “easy” situation is when the “code under test” uses correct and complete libraries.

- So the test code calls the code under test and interprets the output.

The “hard” situation is when the “code under test” uses code that cannot be used in the test

- Because it does not exist yet
- Because it takes too long to run
- Because it might be buggy
- ...

You can create “stubs” – short functions that work enough like the “real code” for testing purposes.

Stub examples

- I wrote a wrong version of `getline` while working on homework 4 on a machine without it.
- Data structures that have data “hard-wired” in.
- “Input” functions that “already know” what they will return
- “Output” functions that do less than in reality
- Code that is wrong but type-checks (to satisfy the compiler)

Integration Testing

When you combine separately tested parts, you need to test that too.

Whitebox vs. blackbox and coverage issues still apply.

Focus is on the interfaces, but may well lead to new unit tests.