CSE 374
Programming Concepts & Tools

Hal Perkins

Fall 2017

Lecture 15 – Testing

Where we are

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

"Test your software or your users will"

Hunt & Thomas

The Pragmatic Programmer

Software design

"There are two ways of constructing a software design:

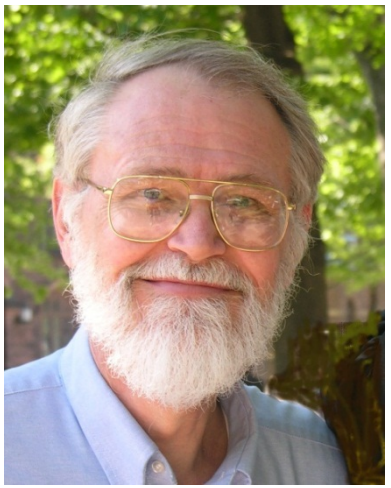
- One way is to make it so simple that there are obviously no deficiencies, and
- the other way is to make it so complicated that there are no obvious deficiencies.

The first method is far more difficult."

Sir C. A. R. Hoare

Debugging

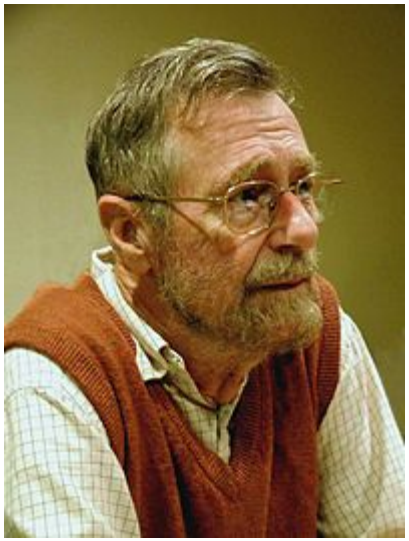
"Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."



Brian Kernighan

Testing

"Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence."



Edsger Dijkstra

1972 Turing Award Lecture

http://userweb.cs.utexas.edu/~EWD/transcriptions/EWD03xx/EWD340.html

Fixing bugs…

- Use languages and tools that make errors impossible when you can
  - Java eliminates a large class of memory bugs
- Don't introduce defects
  - Think, design, analyze – don't write the bugs in the first place!
- Make defects visible
  - Assertions, exceptions (if you have them)
  - Rigorous testing
- Debugging – last resort

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

- 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

How much is enough?

- This code is supposed to compute something resembling C's "a or b" function. How do we test it? How many tests do we need? What kinds of tests should they be?

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

Three 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
  - Cons: can miss internal cases that should be checked
  - Basic examples: remember to try negative numbers; collection: empty, has one element, has many elts.
- *white-box*: test a unit while looking at its implementation (sometimes called "clear box")
  - Pros: can be more efficient, can find the implementation's corner cases
  - Cons: can be biased by implementation assumptions
  - Basic examples: try loop boundaries, "special constants", max values, empty/full data structure…

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. Some useful techniques:

- 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?

Eat 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".

Debugging

- When a test uncovers a problem, need to find the cause and fix it
- Often more art then science, but don't thrash randomly
- Treat debugging as a scientific experiment:
  - Hypothesis: the problem is because …
  - Experiment: design tests to verify hypothesis
  - Not verified? Start over with a new hypothesis
  - Verified? Bug found! Fix it, test it, and add the test that demonstrated the bug to your collection

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.