Software testing (CS258)

Lesson 1: What is testing?

In testing, we are interested in finding failures in software and fixing them. Here we have the test input, the software and the output (i.e. what do we expect to happen). When we run software testing, we have two possibilities:

- The output is “okay” -- we haven't learned a lot
- The output is “bad” -- we have found a bug in our software. It can be caused by:
  - Bug in S.U.T. (Software Under Test)
  - Bug in acceptability test
  - Bugs in specification
  - Bugs in OS, compiler, libraries, hardware

Equivalent tests

- A test mapping a single input to an output
  - Input should represent a "class of inputs", i.e. if the code executes correct for one input, it should work for all input in the class

Creating testable software

- Clean code
- Refactor
- Should always be able to describe what a module does and how it interacts with other code
- No swamp of global variables
- No pointer "soup"
- Modules should have unit tests

Assertions

Check for a property that must be true. Assertions are useful because:

- Make code self-checking, leading to more effective testing
- Make code fail early, i.e. closer to the bug
- Assign blame
- Document assumptions, preconditions, postconditions, invariants

Assertions are common in many large project, for example GCC has ~900 assertions (1 assertion per 110 lines of code). One can disabling assertions in production code, e.g. this is done if we use the optimization flag in Python. However, it is common to keep the assertions in production code. Note that assertions are not a substitute for error handling.

Domains and ranges
Domain is the set of possible inputs. We should tests programs with values sampled from their domain. Range is the set of possible outputs.

Types of testing

- White box
  - Tester knows a lot about the software internals
- Black box
  - The opposite
- Unit testing
  - Testing only a small part of the software
  - Usually white box
- Integration testing
  - Testing multiple software modules (already tested) together
- System testing
  - Does system as a whole work?
  - Usually black box testing
- Differential testing
  - Take same test input and deliver it to two different implementations of SUT
- Stress testing
  - Test systems above their normal usage level
- Random testing
  - Feeding random data into the software

Lesson 2: Coverage testing

Test coverage
Measures the proportion of the program that gets exercised during testing.

Pros
- Objective score
- When coverage is < 100%, we are given a meaningful tasks

Cons
- Not good at finding errors of omission
- Difficult to interpret scores < 100%
- 100% coverage doesn't mean all bugs were found

Coverage metrics

- Function coverage – Has each function in the program been called?
- Statement coverage – Has each statement in the program been executed?
- Branch coverage – Has each branch of each control structure (such as in *if* statements) been executed? For example, given an *if* statement, have both the true and false branches been executed?
- Loop metrics - Execute each loop 0 time, one time and more than once

What does it mean that code doesn't get covered?

- Infeasible code
- Code not worth covering
- Test suite inadequate

How to use coverage

- Use coverage feedback to improve test suite
- Poor coverage -> rethink tests

Lesson 3: Random testing

Random testing involves taking pseudo-random parameters as input to the SUT. It's a good idea to use a seed to ensure that the results are repeatable. Note that in order to do random testing one needs a lot of domain knowledge.

Input validity problem

Thinking about where you want your code to fail (due to the random tests). Example with a browser:

- Random bits -> mainly failures around the protocol handling (i.e. HTTP headers invalid)
- HTTP protocol correct -> mainly fails in HTML parsing
- Etc

Problems with random tests

- Think about which areas you want to be stressed (e.g. stressing programs input validity is usually unnecessary)
- Some programs won't have good validity checks
- Need to think hard about creating good random test generators

Generating random input

- Generative random testing
  - Inputs are created from "scratch"
- Mutation-based random testing
  - Inputs are created by modifying non-randomly generated test cases

Oracles for random testing

Oracles tells the program whether it succeeded or not. Oracles needs to be automated. There are different "levels" of oracles (i.e. how "good" they are):

- Weak oracles
    - Crashes
    - Violation of language rules (e.g. attempting to access list index that doesn't exist)

- Medium Oracle
    - Assertions

- Strong Oracle
    - Different implementation of the same program
    - Older version of the software we are testing
    - Function inverse pair
        - Compress / decompress
        - Save / load
        - Transmit / receive
        - Encode / decode

Lesson 4: Random testing in practice

Why does random testing work?

- Based on weak bug hypothesis
- Developers make same mistakes in coding as in testing
- Huge asymmetry between speed of computers and developers

Why random testing is effective on commercial software?

- Because developers aren't doing it enough

Tuning rules and probabilities

- Start simple -> examine test cases and update accordingly

Can Random Testing Inspire Confidence?

If you have:

- Well-understood API +
- Small code +
- Strong assertions +
- Mature, tuned random tester +
- Good coverage results

Otherwise, you should also use other testing methods.

Tradeoffs in spending time on random testing

Cons

- Input validity can be hard
- Oracles are hard to create
- No stopping criterion
- May find unimportant bugs
- May find same bugs many times
- Can be hard to debug when test case is large and/or makes “no sense”
- Every fuzzer finds different bugs

Pros

- Less tester bias, weaker hypotheses about where bugs are
- Once testing is automated, human cost of testing goes to zero
- Can show bugs that surprises us about our software
- Every fuzzer finds different bugs