Lecture Notes on Testing

15-122: Principles of Imperative Computation

William Lovas

Lecture 19
March 24, 2011

1 Summary

We begin with a discussion of the question, "Why do we test our code?" There are many possible answers, including to increase confidence, to make sure we handle boundary conditions, and to verify that the code meets our expectations. What these ideas all boil down to essentially, though, is that

We test our code to find bugs!

Testing can almost never show complete correctness—there are usually either infinitely many possible inputs, or prohibitively many to test them all—so it is better to think of it as a tool for finding bugs in your code.

When should we test our code? Certainly not just when we discover a bug—testing should be a continual aspect of the development cycle. The methodology of test-driven development goes so far as to require programmers to write test cases before they write any code at all. Tests can serve as a way of understanding the specification of a program, one on par with though complementary to contracts. So perhaps a more general answer is that

We test our code to understand it.

To understand code, we mean to understand its specification, its behavior, and its shortcomings.

2 Useful Terms

black box testing testing only with regard to the specification of the function – its type and its contract – without looking at any of its code. particularly useful when a function has strong contracts that describe its intended behavior very clearly. make creative use of equivalence classes and boundary cases to generate good tests.

glass box testing testing based on looking at the body of a function, trying to come up with tests that exercise it fully. when glass box testing, you should attempt to attain reasonable "code coverage" – your tests should exercise as many possible paths through the program as they can. *a.k.a.,* "white box testing"

boundary cases inputs that are valid according to the specification, but at some context-dependent "boundary". *a.k.a.,* "edge cases".

e.g., `0`, `-1`, `1`, the min `int`, and the max `int`, if the input is an `int`;
e.g., the empty array and any singleton array, if the input is an array;
e.g., the `NULL` pointer, if the input is a pointer.

equivalence class a set of tests such that any test in the set should yield the same behavior. *e.g.,* when testing binary search, the test "search for 5 in $[10, 12]$" is in the same equivalence class as "search for 30 in $[300, 3000]$", "search for 0 in $[1, 2]$", and "search for x in $[y, z]$" for any $x < y < z$.

test-driven development the practice of writing tests before writing code, using the precise language of test cases to clarify a program's specification before development begins.

unit testing testing small pieces of functionality to make sure they behave as intended before using them in a larger program. helps to pin bugs down earlier.

regression testing running old tests on new code, to make sure that previously correct code was not broken while being extended or modified. sometimes fixing new bugs accidentally re-introduces old bugs, and regression testing helps keep development moving in the right direction.

debugging a blank screen having no idea why a program is behaving in an unexpected manner, and making random perturbations to the code in the hopes that they might somehow suddenly fix everything. *a.k.a.,* "programming by random perturbation".

3 Testing Tips

- *Write tests early!* Writing tests can help clarify the spec of a problem and guide your thinking about how to solve it. (They're similar to contracts in this regard—having a hard time writing a contract? Try writing a few tests instead, until the ideas start to gel.) Use lots of small unit tests along the way to building a large program to avoid debugging a blank screen.

- *Run tests often!* Regressions are easy to introduce, and just because your code passed some test once doesn't mean it always will.

- *Don't be afraid to write new code!* Testing often requires the use of auxiliary functions that you might not otherwise need, like pretty-printers, equality-checkers, and convenience constructors. Writing this code can be a fun and cathartic break from more mentally strenuous coding, and it will help you understand and evaluate your code more effectively.

- *Use tests with contracts!* Be sure to compile with `-d` to enable dynamic contract checking when running test code. Contracts and tests complement each other dramatically: if every call to a function checks 10 invariants, then every test case you write for that function is *10 times stronger!*