Unit Testing as Hypothesis Testing

Jonathan Clark

September 19, 2012

5 minutes

You should test your code. Why? To find bugs. Even for seasoned programmers, bugs are an inevitable reality.

Today, we'll take an unconventional view of unit tests: that they're actually a form of hypothesis testing. While hypothesis testing is usually used as a means of making scientific claims in research, for the purposes of this lesson, **hypothesis testing** is just a mechanism for gathering evidence to accept or refute a statement – you won't need to know about hypothesis testing for this class; it's just an analogy.

First, we'll show how to find bugs with unit tests. There, we emphasize that your starting assumption should be that your code is incorrect. Later, we'll see how to use unit tests to narrow down the location of a known bug. In this case, unit tests can help you refine a broad hypothesis such as "there's a bug in my program" to something specific "there's a bug in a one-line function called `safe_add`".

You should walk away with these big points:

- Use unit tests to probe error-prone conditions such as boundary cases
- Use unit tests to aid you in debugging by narrowing down where bugs may be
- Use unit tests to check your contracts
- Use unit tests to verify that failure cases do fail with an error
- Try to write tests that cover many code paths
- Invest time in your tests and you will save time in the long run. Really.

1 Unit Tests Find Bugs by Probing Error-Prone Conditions

15 minutes

Null Hypothesis: My code is incorrect.

You should believe this until you produce overwhelming evidence to the contrary.

To gather evidence against this hypothesis (or for it), you should consider how your code will behave in situations that are simple, extreme, non-obvious, pathological, unlikely, sick, twisted, depraved, despicable, or otherwise nasty. Less colloquially, we refer to such cases as **boundary cases** (or edge cases): inputs that are valid according to the specification, but are in some sense at a minimum, maximum, a or critical point with regard to some context; inputs that are nearby these points may also be considered boundary cases. For example, most programming languages have a minimum and maximum value of their integer type.

Coming up with error-prone cases is itself a creative process in which you should take on a different persona: The Idiot User from Hades or a Meticulous Calculating Demolition Expert. However, you may have less luck coming up with these cases if you start with the assumption that your code is correct.

A good place to start is by examining the restrictions placed on each function. These can come from the function's documentation, the type of each parameter, and contracts. The user rightfully expects that any value within these restrictions will work.

Begin running example: Evil sort. (code included) Question: What cases are likely to fail for a sort? (All of these tests are implemented in `evil_sort_test.c0`)

1. singleton array (this works)
2. empty array (hit! you found a bug)
3. already sorted? (it checks for sorting and does the wrong thing)
4. zero valued elements? (fails. this will turn out to be a comparator bug)

5. negative valued elements? (fails. this will turn out to be a comparator bug)

6. At this point, do we know if our code is correct? (It's not)

7. At this point, do we know if the bug is directly in `evil_sort` or in a function called by `evil_sort`? (No. We'll see how to narrow this down in a bit)

Look over the testing code to understand how it works.

2 Aside: But Don't Contracts Take Care of This?

5 minutes

First, contracts in C_0 are *dynamic* – they're checked at runtime rather than being statically proven true by the compiler.[1] So your contracts are only as reliable as the tests that exercise them. (Later, we'll see that contracts can likewise make your tests more useful as well). You could also consider contracts and unit tests as two sides of the same coin: when you reason with contracts, you're trying to deductively prove your program is correct; when you write unit tests, you're trying to disprove the correctness of your program by counterexample.

Question: Do unit tests allow us to prove/verify the correctness of a program? NO! (A large number of them might be taken as evidence to that effect though.)

Writing good contracts forces you to reason through your code (something you should always be doing anyway) to the point where you should have convinced yourself that you have proven your code to be correct. However, humans are imperfect when it comes to evaluating logical instructions. To claim something is correct does not make it so. Donald Knuth, a father of modern computer science, once said,[2]

> *"Beware of bugs in the above code; I have only proved it correct, not tried it."*

[1] Statically proving program correctness is an active area of research. CMU offers course 15-414 and Prof. Edmund Clarke has published widely in this area.

[2] `http://www-cs-faculty.stanford.edu/~knuth/faq.html`

Automobile engineers design cars to be safe according to known physical laws – they also smash their cars with crash test dummies in them at great cost to make sure they behave as expected. While not cheap, good tests are a worthwhile investment.

3 Unit Tests Refine Hypotheses about a Bug's Location

10 minutes

Starting Hypothesis: There's a bug in my program... somewhere.

So you ran your program via your `main()` function. It didn't give you the output you expected. Now what? Ideally, your unit tests and/or contracts would have caught the bug before we got here (make a mental note to write better unit tests next time around) – but here we are. Here's a rough algorithm for squashing bugs:

1. Add to your list of hypotheses as to where you believe the bug is likely located – these will usually be names of functions.
2. Choose the location where you believe the bug is most likely located. You may have to start with `main()`
3. For that function, note the input values that caused your program to fail. You may need to use `print()` or a stepping debugger to find out.
4. Write a unit test for that function that uses those input values. You should manually determine what the correct, expected output is.
5. If the function fails your test, you can narrow your hypothesis – the bug (or *one* of the bugs) is in that function.
6. Repeat.

Let's say you've narrowed your bug hunt down to function `x()`. But `x()` has a large number of potential child function calls – or all of its child function calls are passing your tests and you're left with several lines where the bug could be. This is probably a good time for refactoring: move blocks of code that are swimming in a larger function to very small functions so that they

can be individually tested. In extreme bug hunts, you may end up with a small number one-line functions.

Continue running example: Evil sort Discussion:

1. Let's say the comparison operator that says whether 2 elements are in order is embedded in evil_sort; how could we use refactoring to help us determine whether the comparator is broken or some other part of evil_sort? (Extract the `is_leq` function – this is in fact already done)
2. Based on the cases that failed before (arrays containing zeros or negative numbers), what tests would you write for the function `is_leq`?
3. `assert(is_leq(1,1) == true);` succeeds.
4. `assert(is_leq(2,3) == true);` succeeds.
5. `assert(is_leq(0,0) == true);` fails.
6. `assert(is_leq(-3,-2) == true);` fails.
7. `assert(is_leq(-2,-3) == false);` fails.
8. Now can we say which function the bug is in? (Even though you still haven't seen the implementation).
9. In practice, use this knowledge to carefully reason through the buggy section of your code!

4 Unit Tests Exercise Contracts

5 minutes

Null Hypothesis: My contracts disallow some inputs that are reasonable according to the spec.

Incorrect contracts on top of an otherwise correct function can cause spurious errors during execution and can cause the user of your code to incorrectly reason about it. Thorough testing can help you spot these issues.

New example: safe_add

1. What boundary cases would you test for safe_add?
2. The following examples will fail due to missing preconditions:
```
assert(safe_add(0,0) == 0);
assert(safe_add(-1,1) == 0);
assert(safe_add(1,-1) == 0);
```
3. What precondition clause do you think might be missing? $(x >= 0 \&\& y <= 0) || (x <= 0 \&\& y >= 0)$;

5 Unit Tests Verify Fail-Fast Behavior

10 minutes

Null Hypothesis: My contracts allow invalid inputs – my code will fail silently when preconditions are not met.

Good error messages dramatically narrow your hypothesis space of Why Things Failed. For example, writing good contracts and evaluating them at runtime can quickly tell you when and where your assumptions have been violated.

However, contracts that don't enforce the conditions that the programmer believes they should can be counter-productive: If you believe that a function's preconditions are being met, you are likely to look elsewhere for a bug first. Bad contracts can misguide your debugging efforts. Therefore, it's important to get them right and test that contracts fail when they should.

Continue example: `safe_add`

1. In what cases do we desire safe_add to fail with a contract violation?
2. `safe_add(2147483647, 1);`
3. `safe_add(-2147483648, -1);`

So far, our tests have used `assert`, but we don't have that option here – we *expect* these cases to fail and annotation failures in C_0 cause programs to exit immediately. So instead, we keep a list of function calls that should fail in a file called `safe_add_fail.coin` and use an I/O redirect to pipe it to

`coin -d`. We can then do a quick manual inspection to make sure that the contracts enforced our preconditions as expected.

6 Testing Glossary

The following glossary is adapted from William Lovas' March 24, 2011 lecture notes:

black box testing testing only with regard to the specification of the function – its type and its contracts – without looking at any of its source code. this can be 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 / white box testing testing based on looking at the body of a function, trying to come up with tests that fully exercise all of its code paths and the boundary cases of each of those code paths.

code coverage the amount of paths through your code that are exercised by your tests.

boundary cases / edge cases inputs that are valid according to the specification, but are in some sense at a minimum, maximum, a or critical point with regard to some context; inputs that are nearby these points may also be considered boundary cases. for example, most programming languages have a minimum and maximum value of their integer type. *Examples:*

- for integers, 0, 1, -1, min int, max int, min int + 1, max int - 1
- for arrays, the empty array, a singleton array, an array having max int elements
- for pointers, a null pointer

corner cases pathological conditions that arise at the intersection of multiple edge cases. for example, when *all* of a function's inputs are set to max int.

equivalence class a set of tests such that any test in the set should yield the same behavior. for example, when tesitng binary search, the test "search for 5 in [10,12]" is in the same equivalence class as "search for 0 in [1,2]".

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. proper regression testing implies that your tests should be *automated*.

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. unit testing helps pin bugs down early and isolate the cause of bugs.

blind debugging 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 suddenly fix everything. also disparigingly referred to as debugging a blank screen. typically, this is counter-productive and results from frustration. best combatted using the testing techniques above.