What Should I Test?

Jack Fox ~nomryg-nilref
FoxyLabs

Abstract

Testing, including unit testing, forms a critical step in software development. This article explores the philosophy and practice of testing in the context of Hoon and Urbit development, particularly as motivated by the development of urQL and Obelisk. Principles and best practices are suggested, as well as a practical example of testing in Hoon.

Contents

What should I test?

Everything. If you don't test it, it doesn't work.

Manuscript submitted for review.

Address author correspondence to ~nomryg-nilref.

1 What does "test everything" even mean?

Everything raises more questions than it answers. However, they fall into two categories, what and how.

No one writes non-trivial software without testing, even if you come to believe that if it builds, it works. The first time you ran your software was a test. Chances are, you ran individual components to see if those worked. Why not take the extra few minutes to record those tests and results? With practice you will get smart about efficiently recording your throw-away tests. And after you read this I hope you will have a new appreciation of why it is worth the extra time to record your tests.

2 What test framework should I use?

Use what is useful and don't get hung-up on a particular framework or methodology. Record tests as you develop. You may think this slows down your development. That's not necessarily a bad thing. You will move forward with confidence that the last thing you worked on really works and you will find yourself thinking through your specification at a deeper level. This is important in a world of mostly incomplete and nebulous specs. What I am driving at is a sort of cousin to Test Driven Development. Think of it as real-time test development.

Your test suite has two important non-obvious functions, as regression tests (OK, this one should be obvious) and as part of your specification. If an existing test starts failing, carefully consider Chesterton's fence (Wikipedia, 2024) before taking the easy road and removing the test.

Regression tests are important not just to prevent unexpectedly breaking functionality, but to allow you to refactor with confidence. Great software is the result of constant refactoring.

3 What is a unit, anyway?

Descending from the hypothetical to the practical, every practical programming language comes with a unit testing environment. In Hoon, this is `/lib/test/hoon` on the `%base` desk. Copy it to `/lib` in your development desk. Here is an article on working with it. The rest of this article assumes a working knowledge of `/lib/test/hoon`.

Don't get hung up on unit testing dogma. A unit is whatever you want it to be. As a subject-oriented programming language, Hoon does not support private properties and functions: every arm and core in your software is accessible for testing. Generally a unit is some self-contained level of functionality, either from the programmer's or the user's point of view.

Here's a more extreme unit test (which some may even prefer to call an integration test):

```
::
::  Build an example bowl manually
++  bowl
  |=  [run=@ud now=@da]
  ^-  bowl:gall
  :*  [~zod ~zod %obelisk `path`(limo `path`/test-agent)]  :: (our src
      [~ ~ ~]                                              :: (wex sup
      [run `@uvJ`(shax run) now [~zod %base ud+run]]       :: (act eny
  ==
```

Now we are ready to test the evolution through time of a database.

```
::
::  time, insert as of 1 second > schema
++  test-time-insert-gt-schema
  =|  run=@ud
  =^  mov1  agent
    %:  ~(on-poke agent (bowl [run ~2000.1.1]))
        %obelisk-action
        !>([%tape-create-db "CREATE DATABASE db1"])
    ==
  =.  run  +(run)
  =^  mov2  agent
```

```
    %:  ~(on-poke agent (bowl [run ~2000.1.2]))
        %obelisk-action
        !>  :+  %tape
                %db1
                "CREATE TABLE db1..my-table (col1 @t) PRIMARY KEY (col
                "AS OF ~2023.7.9..22.35.35..7e90"
    ==
  =.  run  +(run)
  =^  mov3  agent
    %:  ~(on-poke agent (bowl [run ~2023.7.9..22.35.35..7e90]))
        %obelisk-action
        !>  :+  %tape
                %db1
                "INSERT INTO db1..my-table (col1) VALUES ('cord') ".
                "AS OF ~2023.7.9..22.35.36..7e90"
    ==
  =+  !<(=state on-save:agent)
  ;:  weld
  %+  expect-eq
    !>  :-  %results
          :~  [%result-da 'data time' ~2023.7.9..22.35.36..7e90]
              [%result-ud 'row count' 1]
              ==
    !>  ->+>+>.mov3
  %+  expect-eq
    !>  db-time-insert-tbl
    !>  databases.state
  ==
```

Under the covers the Obelisk engine calls out to the urQL parser to create AST commands from the user-created scripts. The `AS OF` clause overrides the time passed in the bowl. Finally we weld together two results and check them against what we expect. The first check, expect-eq, is of the metadata returned by the last command. The second check is the final database state after the evolution.

Unit testing purism insists on specifically tailoring each test to one specific case of potential failure. This approach tends to be pedantic and frequently we pragmatically sneak in multiple independent tests in one test bundle, as in Listing 1.

As you can see, there was quite a bit tested in `test-alter-`

Listing 1: Multiple tests in one bundle

```
:: common things to test
:: 1) basic command works producing AST object
:: 2) multiple ASTs
:: 3) all keywords are case ambivalent
:: 4) all names follow rules for faces
:: 5) all qualifier combinations work
::
:: alter index
::
:: tests 1, 2, 3, 5
::       extra whitespace characters
::       multiple command script:
::         alter index... db.ns.index db.ns.table columns action %disa
::         alter index db..index db..table one column
::         action %rebuild
++  test-alter-index-1
  =/  expected1
    :*  %alter-index
        :*  %qualified-object
            ship=~ database='db'
            namespace='ns' name='my-index'
        ==
        :*  %qualified-object
            ship=~ database='db'
            namespace='ns' name='table'
        ==
        :~  :*  %ordered-column
                name='col1'
                is-ascending=%.y
            ==
            :*  %ordered-column
                name='col2'
                is-ascending=%.n
            ==
            :*  %ordered-column
                name='col3'
                is-ascending=%.y
        ==  ==
        %disable
        ==

  =/  expected2
    :*  %alter-index
        :*  %qualified-object
            ship=~ database='db'
            namespace='dbo' name='my-index'
        ==
```

`index-1` above. Things like whitespace and mixed-case labels are tedious to exhaustively test and well-suited to property based testing software like `%quiz` (see below).

4 Proof by Induction

We were intentionally dismissive of testing methodologies above because they are mere conventions. There is however some actual theory we can leverage in figuring out what and how to test.[1]

Peano arithmetic is a better lay programmer's introduction to proof by induction. There exists (or perhaps *does not exist*) a special concept, zero. Playing fast and loose with classical logic, "ex nihil sequitur quodlibet", from nothing (more commonly a falsehood, or a contradiction) follows everything (i.e. anything).[2] Zero is not only an integer; rather, in terms of Nock-based programming, ~ is not only the beginning of counting,[3] but the nothing of every inductive type, most importantly trees and lists. When speaking to programmers, Peano tells us there is a universal nothing ~ and there is a function called successor Succ, which produces some next thing from a previous thing (or lack of thing). Applying Succ to nothing, $\mathrm{Succ}(0)$, gives us the first thing. Applying Succ to the first thing, $\mathrm{Succ}(\mathrm{Succ}(0))$ gives us the second thing, and so on.

Induction appears in unexpected places—think zero-length strings, which do not appear to involve ~ at all. For instance, the Hoon type `unit` is also an inductive type: it is literally ei-

[1] **TODO** provides a technical exposition of mathematical induction. You can skip it because we will explain it non-rigorously for lay programmers. We also quibble about the inclusion of zero as a natural number. While old school maths started the natural numbers with "1", but computer science has since infected maths. Have you ever seen zero of anything? No, it is not natural at all. Zero is a very, very special number.2 It's an abstraction that does not map to anything in the physical world.

[2] The "principle of explosion" rigorously follows from use of disjunctive syllogisms. Here, we jocosely indicate that by proceeding from zero inductively we can demonstrate desired properties of testing.

[3] Yarvin opted to invert `true` and `false` (e.g. `urbit/vere` commit `a8c1a799`, 2014-11-04), meaning that Nock loobeans do not align inductively with other inductive types.

Listing 2: Sequent tests

```
::
::  +contains
++  test-contains-00
  %+  expect-eq
    !>  %.n
    !>  (contains `(list)`~ "yep")
++  test-contains-01
  %+  expect-eq
    !>  %.y
    !>  (contains `(list @)`~[1] 1)
++  test-contains-example-00
  %+  expect-eq
    !>  %.y
    !>  (contains `(list tape)`~["nope" "yep"] "yep")
```

ther nothing or something. Knowing now what to look for, see where you can find induction in your own code.

What does induction have to do with testing, however? For inductive types—and simple functions over inductive types—the software author only has to prove, or test, two cases: the case for zero and the case for the successor value after zero. For example, Listing 2 depicts tests for a gate over an inductive type (`list`) in `/lib/seq` (`~nomryg-nilref @jackfoxy` (2024), currently distributed via `%yard` (*Yard: A Developer Commons* 2024)).

"Hey", you say—"that's more than two tests." That's right: two is the bare minimum of required tests, and only applies to the simplest units of inductive testing. In this case the first two tests suffice, but minimal tests frequently make for unhelpful documentation examples. We heartily recommend providing interesting examples in your documentation and including those examples in your test suite. You don't want users to struggle with examples that don't work, or worse, don't even build. More tests don't hurt anything; the computer doesn't get tired.

Another reason for additional tests is taking a page out of

"white box" or "gray box" testing. If you know that there is special logic for the first successor case, you need to test the first case independently as well as a subsequent successor. If you are the programmer and the tester you should approach all of your testing from this perspective. You might even see how to make your program simpler.

4.1 The First Cartesian Explosion

Through the mathematics of currying programmatic functions, we can have input arguments of multiple inductive types.[4] This results in the minimum number of tests being the number of inductive input elements squared, starting with all elements set to ~ and so forth. Listing 3 depicts a series of tests for the `append` gate subject to this n^2 explosion.

In this situation, the required number of tests may grow exponentially but in most practical cases remains a relatively small finite number.

4.2 The Second Cartesian Explosion

Input argument interactions are not the only source of combinatorial explosion in testing. Imagine that your function (gate) is a black box. You start submitting random input to figure out what the underlying algorithm is. However, it turns out that you discover that there is not one consistent algorithm for all inputs. Some values or ranges of values behave differently from others. (Unexpected UTF-8 whitespace characters are notorious for revealing head-scratching bugs—so imagine all possible inputs over complex XML.) You could model this behavior as multiple inductive types making for an even bigger cartesian explosion of inputs to test. It is no longer practical to construct all the tests required by our theory. The number is still finite, but impractically large. What is to be done?

[4] While most programming languages handle multiple function inputs via currying, Hoon gates always accept a single noun, which can be a cell. So in this case the currying is not even theoretical.

Listing 3: Multiple single tests

```
::
::  +append
++  test-append-00
  %+  expect-eq
    !>  ~
    !>  (append ~ ~)
++  test-append-01
  %+  expect-eq
    !>  ~[1]
    !>  (append ~[1] `(list)`~)
++  test-append-02
  %+  expect-eq
    !>  ~[1]
    !>  (append ~ ~[1])
++  test-append-03
  %+  expect-eq
    !>  ~[1 2]
    !>  (append ~[1] ~[2])
```

There is no general solution. Complete code coverage with tests is a start, inductively testing over each clause in a unit. This is time-consuming and requires thinking deeply about the code and its structure.

Another approach is to favor so-called edge case testing, in which you test the boundaries of the input space. This is a good idea, but it is not a complete solution. It is not always clear what the edge is, and it is not always clear that the edge is the same for all inductive types. In the case of testing a string function, should an edge be the empty string, a string with one character, or a string with two characters?

An automated testing solution called property-based testing may be applicable in these cases. The idea of property-based testing is to develop and instantiate invariant properties of the code and let software generate random inputs. The software runs the random inputs and tests the outputs against the invariant properties. An architecture for this approach was originally developed for Haskell and has since ported to many

other languages, including Hoon. `%quiz` is a well-documented Hoon implementation (`~bithex-topnym @hjorthjort`, 2023). Once again, this solution requires some deep thinking about the code and its specification.

From our experience with the F# implementation of property-based testing, we expect that you will need to boost the number of random input permutations beyond the default of 100 to get the kind of coverage that is "reasonably" exhaustive. We found 10,000 to be frequently adequate. (Since the inputs are randomly generated each run, however, a property test of production code may fail when nothing has changed. Then think about how to explain to your boss that your tests are not deterministic.)

Lastly, whenever you fix a bona fide production bug (or one that a framework like `%quiz` discovered), add a test case to address that circumstance. This not only provides the standard for when the bug has been fixed, but protects against regressing to the prior behavior (thus, a "regression test"). Congratulations, you have just refined your specification.

5 Testing Failure

In our experience failure modes are the most overlooked part of sofware development. It starts with passing insufficient, or no information from the programmed points of failure. So even before testing failure modes make sure you distinguish (i.e. make unique) each message from every point of failure and include any and all relevant information available for debugging.

```
~|("cannot add duplicate key: {<row-key>}" !!)
```

The standard `/lib/test` library, as of this writing, can test for failure but not for an expected message. A modified testing arm (Listing 4) can be included in a testing thread to facilitate this kind of testing. Listing 5 shows a test for an expected error message in the urQL database engine which uses this functionality to verify that the correct error message is raised on crash.

Listing 4: Testing for expected error message

```
::
::  +expect-fail-message
++  expect-fail-message
  |=  [msg=@t a=(trap)]
  ^-  tang
  =/  b  (mule a)
  ?-  -.b
    %|  |^
        =/  =tang  (flatten +.b)
        ?:  ?=(^ (find (trip msg) tang))
          ~
        ['expected error message - not found' ~]
        ++  flatten
          |=  tang=(list tank)
          =|  res=tape
          |-  ^-  tape
          ?~  tang  res
          %=  $
            tang  t.tang
            res   (weld ~(ram re i.tang) res)
          ==
        --
    %&  ['expected failure - succeeded' ~]
  ==
```

Listing 5: urQL tests

```
::
:: fail on dup rows
++  test-fail-insert-dup-rows
  =|  run=@ud
  =/  my-insert
    "INSERT INTO db1..my-table (col1, col2, col3)  ".
    "VALUES ('cord',~zod,20) ('Default',Default, 0)"
  %+  expect-fail-message
    'cannot add duplicate key:'
  |.  %-  process-cmds
      :+  gen3-dbs  :: <- one key already exists
        (bowl [run ~2031.1.1])
      (parse:parse(default-database 'db1') my-insert)
```

6 Conclusion

Keep the following principles in mind when producing and evaluating code as a software developer.

1. Strive to make the collection of units of testing exhaustive, both primitive units of code and units of work from the user perspective.
2. Test inductively wherever possible.
3. Test from a white or gray box perspective.
4. Test the failure modes.
5. Test all the examples in your documentation.
6. In a world lacking documentation tests may be the only real specification.[5]
7. Regression tests are the key to refactoring with confidence. Beautiful code comes from refactoring.

[5] Cf. the definition by Feathers, p. xvi that "legacy code is simply code without tests." The entire text may thus be commended as a thorough guide to testing despite its name.

References

~bithex-topnym @hjorthjort (Aug. 14, 2023). *Quiz: A randomized property testing library for Urbit*. URL: https://github.com/hjorthjort/quiz (visited on 02/07/2024).

Feathers, Michael C. (2005). *Working Effectively with Legacy Code*. Prentice Hall. ISBN: 978-0-13-117705-5.

~nomryg-nilref @jackfoxy (2024). *Sequent: A library of Hoon list functions for mortal developers*. GitHub.

Wikipedia (2024). *G. K. Chesterton, Chesterton's Fence*. URL: https://en.wikipedia.org/wiki/G._K._Chesterton (visited on 02/07/2024).

Yard: A Developer Commons (2024). GitHub.