https://medium.com/javascript-scene/what-every-unit-test-needs-f6cd34d9836d#.xdvvtym3t

## Most Developers Don't Know How to Test

Every developer knows we should write unit tests in order to prevent defects from being deployed to production.

What most developers don't know are the essential ingredients of **every unit test**. I can't begin to count the number of times I've seen a unit test fail, only to investigate and discover that I have absolutely no idea what feature the developer was trying to test, let alone how it went wrong or why it matters.

In a recent project of mine, we let a gigantic swath of unit tests enter the test suite with **absolutely no description whatsoever** of the test's purpose. We have a great team, so I let my guard down. The result? We still have a ton of unit tests that only the author can really make sense of.

Luckily, we're completely redesigning the API, and we're going to throw the entire suite away and start from scratch—otherwise, this would be **priority #1** on my fix list.

Don't let this happen to you.

## Why Bother with Test Discipline?

Your tests are your first and best line of defense against software defects. Your tests are more important than linting & static analysis (which can only find a subclass of errors, not problems with your actual program logic). Tests are as important as the implementation itself (all that matters is that the code meets the requirement—how it's implemented doesn't matter at all unless it's implemented poorly).

Unit tests combine many features that make them your secret weapon to application success:

1. **Design aid:** Writing tests first gives you a clearer perspective on the ideal API design.
2. **Feature documentation (for developers):** Test descriptions enshrine in code every implemented feature requirement.
3. **Test your developer understanding:** Does the developer understand the problem enough to articulate in code all critical component requirements?

4. **Quality Assurance:** Manual QA is error prone. In my experience, it's impossible for a developer to remember all features that need testing after making a change to refactor, add new features, or remove features.

5. **Continuous Delivery Aid:** Automated QA affords the opportunity to automatically prevent broken builds from being deployed to production.

Unit tests don't need to be twisted or manipulated to serve all of those broad-ranging goals. Rather, it is in the essential nature of a unit test to satisfy all of those needs. These benefits are all side-effects of a well-written test suite with good coverage.

### The Science of TDD

The evidence says:

- **TDD can reduce bug density.**
- **TDD can encourage more modular designs** (enhancing software agility/team velocity).
- **TDD can reduce code complexity.**

**Says science:** There is *significant empirical evidence that TDD works**.

## Write Tests First

Studies from Microsoft Research, IBM, and Springer tested the efficacy of test-first vs test-after methodologies and consistently found that a test-first process produces better results than adding tests later. It is resoundingly clear: Before you implement, write the test.

. . .

*Before you implement,*
*write the test.*

. . .

## What's in a Good Unit Test?

OK, so TDD works. Write tests first. Be more disciplined. Trust the process… We get it. But **how do you write a good unit test?**

We're going to look at a very simple example from a real project to explore the process: The `*compose()*` function from the Stamp Specification.

We're going to use tape for the tests because of its crystal clarity and essential simplicity.

Before we can answer how to write a good unit test, first we have to understand how unit tests are used:

- **Design aid:** written during design phase, *prior to implementation*.
- **Feature documentation & test of developer understanding:** The test should provide a *clear description of the feature being tested.*
- **QA/Continuous Delivery:** The tests should halt the delivery pipeline on failure and *produce a good bug report when they fail.*

## The Unit Test as a Bug Report

When a test fails, that test failure report is often your first and best clue about exactly what went wrong—the secret to tracking down a root cause quickly is knowing where to start looking. That process is made much easier when you have a really clear bug report.

. . .

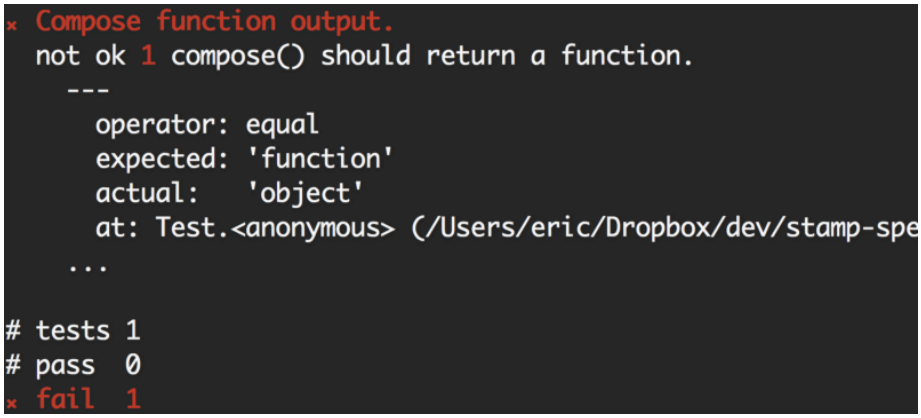
*A failing test should read*
*like a high-quality bug report.*

. . .

### What's in a good test failure bug report?

1. **What were you testing?**
2. **What should it do?**
3. **What was the output (actual behavior)?**
4. **What was the expected output (expected behavior)?**

. . .




Example of a good failure report.

. . .

## Start by answering "what are you testing?":

- **What component aspect** are you testing?
- **What should the feature do?** What specific behavior requirement are you testing?

The `*compose()*` function takes any number of stamps (composable factory functions) and produces a new stamp.

To write this test, we're going to work backwards from the end goal of any single test: To test a specific behavior requirement. In order for this test to pass, what specific behavior must the code produce?

## What should the feature do?

I like to start by writing a string. Not assigned to anything. Not passed into any function. Just a clear focus on a specific requirement that the component must satisfy. In this case, we'll start with the fact that the `*compose()*` function should return a function.

A simple, testable requirement:

```
'compose() should return a function.'
```

Now we'll skip some stuff and flesh out the rest of the test. This string is serving as our goal. Stating it beforehand helps us keep our eye on the prize.

## What component aspect are we testing?

What you mean by “component aspect” will vary from test to test, depending on the granularity required to provide adequate coverage of the component under test.

In this case, we’re going to test the return type of the `*compose()*` function to make sure it returns the right kind of thing, as opposed to `*undefined*` or nothing at all because it throws when you run it.

Let’s translate this question into test code. The answer goes into the test description. This step is also where we’ll make our function call and pass the callback function that the test runner will invoke when the tests run:

```
test('<What component aspect are we testing?>', assert => {
});
```

In this case, we’re testing the output of the compose function:

```
test('Compose function output type.', assert => {
});
```

And of course we still need our first description. It goes inside the callback function:

```
test('Compose function output type.', assert => {
  'compose() should return a function.'
});
```

# What is the Output (Expected and Actual)?

`*equal()*` is my favorite assertion. If the only available assertion in every test suite was `*equal()*`, almost every test suite in the world would be better for it. Why?

Because `*equal()*`, by nature answers **the two most important questions every unit test must answer**, but most don’t:

- **What is the actual output?**
- **What is the expected output?**

If you finish a test without answering those two questions, you don't have a real unit test. You have a sloppy, half-baked test.

If you take only one thing from this article, let it be this:

. . .

> *Equal is your new default assertion.*
> *It is the staple of every good test suite.*

. . .

All those fancy assertion libraries with hundreds of different fancy assertions are **destroying the quality of your tests.**

## A Challenge

Want to get much better at writing unit tests? For the next week, try writing every single assertion using `*equal()*` or `*deepEqual()*`, or their equivalents in your assertion library of choice. Don't worry about the quality impact on your suite. My money says that the exercise will *improve it dramatically.*

What does this look like in code?

```
const actual = '<what is the actual output?>';
const expected = '<what is the expected output?>';
```

The first question really does double duty in a test failure. By answering the question, your code also answers another:

```
const actual = '<how is the test reproduced?>';
```

It's important to note that **the `*actual*` value must be produced by exercising some of the component's public API.** Otherwise, the test has no value. I've seen test suites that are so overwhelmed with mocks and stubs

and bells and whistles that some of the tests never exercised any of the code supposedly being tested.

Let's return to the example:

```
const actual = typeof compose();
const expected = 'function';
```

You could build an assertion without specifically assigning values to variables called `*actual*` and `*expected*`, but I recently started to specifically assign values to variables called `*actual*` and `*expected*` **in every test** and found that **it made my tests easier to read.**

See how it clarifies the assertion?

```
assert.equal(actual, expected,
  'compose() should return a function.');
```

It separates the "how" from the "what" in the test body.

- Want to know **how we got the results?** Look at the *variable assignments*.
- Want to know **what we're testing for?** Look at the *assertion's description*.

The result is that the test itself reads as easily as a high quality bug report.

**Let's look at the whole thing in context:**

```
import test from 'tape';
import compose from '../source/compose';

test('Compose function output type', assert => {
  const actual = typeof compose();
  const expected = 'function';

  assert.equal(actual, expected,
    'compose() should return a function.');
```

Next time you write a test, remember to answer all the questions:

1. **What are you testing?**
2. **What should it do?**
3. **What is the actual output?**
4. **What is the expected output?**
5. **How can the test be reproduced?**

The last question is answered by the code used to derive the `*actual*` value.

## A Unit Test Template:

```
import test from 'tape';

// For each unit test you write,
// answer these questions:
test('What component aspect are you testing?', assert => {
  const actual = 'What is the actual output?';
  const expected = 'What is the expected output?';

  assert.equal(actual, expected,
    'What should the feature do?');
```

There's a whole lot more to using unit tests well, but knowing how to write a good test goes a long way.

. . .

. . .

```
let tern = x=> isNaN(x) ? ( x!==x ? 'NaN' : (undefined===x ? 'undefined' :
(isNaN(x) ? 'NaN' : 'qqq' ) ) ) : (x === 0? 'number' : ( isNaN(parseInt(x)) ?
'NaN' : 'number' ) );


module.exports = tern;
```

```
var test = require('tape'),
    tern = require('./tern');


test('ternary function', assert => {
    const actual = typeof tern(''),
          expected = 'NaN';

    assert.equal(actual, expected, 'tern() should return NaN');
    //провал - почему?
    //потому что мы ожидаем NaN по смыслу
    //а функция tern возвращает строку по факту
    //т.е. мы видим в консоли NaN, но это - строка, а не значение NaN

    assert.end();
});
```

```
 $ node test1.js
TAP version 13
# ternary function
not ok 1 tern() should return NaN
  ---
    operator: equal
    expected: 'NaN'
    actual:   'string'
    at: Test.assert (/Users/eliasgoss/Dropbox/p_r_o_j_e_c_t_s/js_testing/unit/te
st1.js:11:12)
  ...

1..1
# tests 1
# pass  0
# fail  1

➜ eliasgoss@ ~/Dropbox/p_r_o_j_e_c_t_s/js_testing/unit
```