Tip 32: Write Functions for Testability

In this tip, you'll learn about writing functions for testability and dependency injection.

We'll cover the following

- Testing
- Writing testable code
  - Imported code problems
- Dependency injection

I had a literature professor who said that classes about writing don't include enough reading and classes about reading don't include enough writing. The same is true of code and tests: Books about code don't talk enough about testing and books about testing don't talk enough about composing code.

Time to fix that.

Testing

Testing is important. If you don't do it, you should. It makes your code easier to refactor. It makes legacy code much easier to understand. And it generally results in cleaner, less buggy applications.

Most developers agree with this. *Why then is testing neglected?*

It's simple. Writing tests is hard. Or more accurately, many developers think writing tests is hard because they try to fit tests onto their existing code. And their existing code is tightly coupled with external dependencies.

Code that's hard to test is often unnecessarily complex. Instead of struggling to make tests for your code, you should focus on writing code that is testable. Your code will improve, your tests will be easier to write, and the user experience will be identical. There's nothing to lose.

If you're new to testing, check the documentation for one of three popular testing frameworks—jasmine, mocha, or jest—for some quick pointers. You can also check frameworks — jasmine, mocha, or jest — for some quick pointers. You can also check

out the testing of the code for this course in the end, which has near 100% code coverage using mocha as the test runner.

To get the most out of this tip, you should know the basics of `describe()` and `it()` functions along with expectations.

Writing testable code

Now, how do you write testable code? Here's a function that looks simple but has some subtle complexity.

```
import {getTaxInformation} from "./taxService.js";

function formatPrice(user, { price, location }) {
    const rate = getTaxInformation(location);
    const taxes = rate ? `plus ${price * rate} in taxes.` : 'plus tax.';

    return `${user} your total is: ${price} ${taxes}`;
}

const item = { price: 30, location: 'Oklahoma' };
const user = 'Aaron Cometbus';
console.log(formatPrice(user,item));

export { formatPrice };
```

You may be wondering, how can this be complex? All it does is compute some tax information from a price and combines it with a user to create a string.

The testing difficulty begins when you call an outside function on **line 4**. Notice that you're importing that function at the top of the file. You'll learn more about importing functions in Tip 47, Isolate Functionality with Import and Export, but for now, all you need to know is that you're getting something from outside the file.

Imported code problems

The problem with using imported code directly is that the function is now tightly coupled with the imported function. You can't run `formatPrice()` without executing `getTaxInformation()`. And because the `getTaxInformation()` function will likely need to hit an external service or a config file, you're now tightly coupled to network communication. This means that if you run a test, the test will also have

to access the API. Now your test is dependent on network access, response time, and so on. Again, this is a big problem. You're just trying to build a string.
To avoid the problem, you can create mocks that intercept imports and explicitly set a return value. Here's what a test would look like for the current function.

```
import expect from 'expect';
import sinon from 'sinon';
import * as taxService from ''./taxService';
import { formatPrice } from './problem';

describe('format price', () => {
    let taxStub;

    beforeEach(() => {
        taxStub = sinon.stub(taxService, 'getTaxInformation');
    });

    afterEach(() => {
        taxStub.restore();
    });

    it('should return plus tax if no tax info', () => {
        taxStub.returns(null);
        const item = { price: 30, location: 'Oklahoma' };
        const user = 'Aaron Cometbus';
        const message = formatPrice(user, item);
        const expectedMessage = 'Aaron Cometbus your total is: 30 plus tax.';
        expect(message).toEqual(expectedMessage);
    });

    it('should return plus tax information', () => {
        taxStub.returns(0.1);

        const item = { price: 30, location: 'Oklahoma' };
        const user = 'Aaron Cometbus';
        const message = formatPrice(user, item);
        const expectedMessage = 'Aaron Cometbus your total is: plus $3 in taxes.';
        expect(message).toEqual(expectedMessage);
    });
});
```

The tricky part begins on **line 10**. You're creating a stub that overrides the original `getTaxInformation()` function with a simple return value.

When you create a stub, you're bypassing the imported code and declaring what the output would be without running the actual code. The upside is that now you don't have to worry about any external dependencies. The downside is that you constantly have to set and reset the return value in every assertion. See **line 19** for an example.

Finally, after the test suite is over, you have to restore the code to use the original method. You do this in the `afterEach()` method on **line 15**. Restoring the code is a crucial step. By hijacking the code in this test suite, you've hijacked it for all tests unless you restore it.

Dependency injection

I once had a test suite that was tightly coupled and used a lot of stubs. Everything was working until I changed the location of a file. All of a sudden, the tests ran in a different order and lots of tests started failing. I thought I had accurately restored all the stubs, but it was an illusion. The only reason the test passed was because they ran in a specific order.

Don't be fooled by the shortness of the test suite. Tests that require a lot of external helpers, such as spies, mocks, and stubs, are a clue that your code is complex and may be tightly coupled. You should simplify your code.

Fortunately, the fix for tightly coupled code is fairly simple. You simply pass in your external functions as arguments. Passing in dependencies as arguments is called **dependency injection**.

Stubs, Mocks, Spies

To decouple your code, pass `getTaxInformation()` as an argument. You don't need to change anything else in your code.

```
function formatPrice(user, { price, location }, getTaxInformation) {
    const rate = getTaxInformation(location);
    const taxes = rate ? `plus ${price * rate} in taxes.` : 'plus tax.';
    return `${user} your total is: ${price} ${taxes}`;
}

const item = { price: 30, location: 'Oklahoma' };
const user = 'Aaron Cometbus';
console.log(formatPrice(user,item,() => null));
console.log(formatPrice(user, item, () => 0.1));

export { formatPrice };
```

Now that you're using *dependency injection*, you don't need stubs. When you write your tests, you don't need to bypass an import. Instead, you pass a simple function that returns the value you want. It's a lot like stubbing but without any external dependencies. Your function now takes inputs, including another function, and returns outputs. Remember, you aren't testing `getTaxInformation()`. You're testing that `formatPrice()`, given certain inputs, will return a certain result.

Here's your test:

```
import expect from 'expect';
import { formatPrice } from './test';

describe('format price', () => {
    it('should return plus tax if no tax info', () => {
        const item = { price: 30, location: 'Oklahoma' };
        const user = 'Aaron Cometbus';
        const message = formatPrice(user, item, () => null);
        expect(message).toEqual('Aaron Cometbus your total is: 30 plus tax.');
    });
    it('should return plus tax information', () => {
        const item = { price: 30, location: 'Oklahoma' };
        const user = 'Aaron Cometbus';
        const message = formatPrice(user, item, () => 0.1);
        expect(message).toEqual('Aaron Cometbus your total is: 30 plus $3 in taxes.');
    });
});
```

Notice that you require nothing except the function you're testing and the expect library. The tests are much easier to write, and they do a better job of getting your code down to a single responsibility.

You may argue that dependency injection didn't solve the problem—*it moved the problem to another function*.

That's true. There are going to be some side effects, some input/output, in your code. The trick to writing testable code is to get that in as few places as possible.

For example, you can move all your AJAX calls into a service. Then, when you need to use them in a function, you can inject a service that's easy to test rather than trying to mock AJAX responses (which is very difficult).

The important thing to know is that there's a perception that writing tests is hard. That's just not true. If a test is hard to write, spend time rethinking your code. If your code isn't easy to test, you should change your code, not your tests.

And don't get frustrated when you encounter other problems. Tightly coupled code is just one form of complexity. There are plenty of other code smells—code that's technically correct but doesn't seem very clear— that sneak into tests. Joshua Mock wrote a good article on some of the other problems of testing JavaScript, and it's worth reading to learn more.

The best thing you can do is to start writing tests today. If you need more examples, check out the testing for this course's codes below. It has nearly 100% test coverage and has a variety of tests (including some with mocks and spies). If you want to learn more, check out Test Driving JavaScript Applications.

Terminal

In the next tip, we'll get back into the details of writing functions by further exploring arrow functions.