## **MORE TESTING**

### **Introduction**

An important basic concept in testing is isolation. You should only test one method at a time, and your tests for one function should not depend upon an external function behaving correctly - especially if that function is being tested elsewhere. The main reason for this is that when your tests fail, you want to be able to narrow down the cause of this failure as quickly as possible. If you have a test that depends on several functions, it can be hard to tell exactly what is going wrong.

### **Pure Functions**

There are *many* benefits to using TDD when you write your code. One of the biggest benefits is less obvious at first - it helps you to write better code. If you look back at some of your early projects you will probably notice how *tightly coupled* everything is. All of your functions include references to functions in other parts of your code, and the whole thing is *filled* with DOM methods or console.log().

Tightly coupled code is hard to test! Imagine trying to write tests for a function like this:

function guessingGame() { const magicNumber = 22; const guess = prompt('guess a number between 1 and 100!'); if (guess > magicNumber) { alert('YOUR GUESS IS TOO BIG'); } else if (guess < magicNumber) { alert('YOUR GUESS IS TOO SMALL'); } else if (guess == magicNumber) { alert('YOU DID IT! 🎉'); } }

Making this testable requires us to split up all the different things that are happening. First, we do not need to test the functions prompt and alert because they are built into the browser. They are external to our program and whoever wrote them has already tested them. What we *do* need to test is the number logic, which is much easier if we untangle it from the other functions:

function evaluateGuess(magicNumber, guess) { if (guess > magicNumber) { return 'YOUR GUESS IS TOO BIG'; } else if (guess < magicNumber) { return 'YOUR GUESS IS TOO SMALL'; } else if (guess == magicNumber) { return 'YOU DID IT! 🎉'; } } function guessingGame() { const magicNumber = 22; const guess = prompt('guess a number between 1 and 100!'); const message = evaluateGuess(magicNumber, guess); alert(message); } guessingGame();

In this example, the only thing we really need to test is the evaluateGuess function, which is much easier to test because it has a clear input and output and doesn’t call any external functions. This implementation is *much* nicer as well because it’s much easier to extend. If we wanted to switch out the prompt and alerts for methods that manipulate the DOM we can do that more simply now and if we want to make our game more advanced by letting the user make multiple guesses, that is also easier.

If we had written this program with TDD it is very likely that it would have looked more like the second example to begin with. Test driven development encourages better program architecture because it encourages you to write *Pure Functions*.

* Read [this quick article](https://medium.com/@jamesjefferyuk/javascript-what-are-pure-functions-4d4d5392d49c) about the value of ‘Pure Functions’.

### **Mocking**

There are two solutions to the ‘tightly coupled code’ problem. The first, and best option is to simply remove those dependencies from your code as we did above, but that is simply not always possible. The second option is mocking - writing “fake” versions of a function that always behaves *exactly* how you want. For example, if you’re testing a function that gets information from a DOM input, you really don’t want to have to set up a webpage and dynamically insert something into the input just to run your tests. With a mock function, you could just create a fake version of the input-grabbing function that always returns a specific value and use THAT in your test.

### **Assignment**

1. If you haven’t already, watch the ‘mocking’ videos from [this series](https://www.youtube.com/watch?v=3PjdxjWK0F0).
2. Too much mocking can be a bad thing. It *is* sometimes necessary, but if you have to set up an elaborate system of mocks to test any bit of your code, that means your code is too tightly coupled. These two articles ([one](https://medium.com/javascript-scene/why-i-use-tape-instead-of-mocha-so-should-you-6aa105d8eaf4) and [two](https://medium.com/javascript-scene/mocking-is-a-code-smell-944a70c90a6a)) might be a little extreme, but they contain several really good points about program architecture and testing.
3. Now that you have some practice and context for TDD, [this section](https://facebook.github.io/jest/docs/en/setup-teardown.html#content) of the Jest docs will probably make good sense to you.
4. Jest includes some *really* handy mocking functions. Read about them in the [official docs](https://facebook.github.io/jest/docs/en/mock-functions.html#content)
5. And finally, if you wish, you can add Jest to your webpack setup. Read about that process [here](https://facebook.github.io/jest/docs/en/webpack.html#content).