Extracted from:

Test-Driving JavaScript Applications

Rapid, Confident, Maintainable Code

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Test-Driving JavaScript Applications

Rapid, Confident, Maintainable Code

Venkat Subramaniam

Edited by Jacquelyn Carter

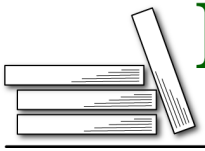
Test-Driving JavaScript Applications

Rapid, Confident, Maintainable Code

Venkat Subramaniam

The Pragmatic Bookshelf

Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

The team that produced this book includes:

Jacquelyn Carter (editor)
Potomac Indexing, LLC (index)
Liz Welch (copyedit)
Gilson Graphics (layout)
Janet Furlow (producer)

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-174-2
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—October 2016

Separate Concerns

The test list now has one incomplete test, "printReport prints the results in sorted order," which seemed like a good test at the end of the strategic design. But looking at the code we've designed so far, that appears like a giant step. We need to break it into smaller functions. Print and sort are two separate concerns—a good design keeps separate concerns apart. Also, we don't want printReport to write to the console; it's better to send the data to the caller so they can decide where to print it. Let's jot down the tests that just emerged:

- ...
- ~~printReport prints the results in sorted order~~
- printReport sends price, errors once all responses arrive
- printReport does not send before all responses arrive
- printReport sorts prices based on the tickers
- printReport sorts errors based on the tickers

Design printReport

We'll implement the first two tests and then look at the sort feature after that.

testnode/stockfetch/test/stockfetch-test.js

```
it('printReport should send price, errors once all responses arrive',
  function() {
  stockfetch.prices = { 'GOOG': 12.34 };
  stockfetch.errors = { 'AAPL': 'error' };
  stockfetch.tickersCount = 2;

  var callbackMock =
    sandbox.mock(stockfetch)
           .expects('reportCallback')
           .withArgs([['GOOG', 12.34]], [['AAPL', 'error']]);

  stockfetch.printReport();
  callbackMock.verify();
});

it('printReport should not send before all responses arrive', function() {
  stockfetch.prices = { 'GOOG': 12.34 };
  stockfetch.errors = { 'AAPL': 'error' };
  stockfetch.tickersCount = 3;

  var callbackMock = sandbox.mock(stockfetch)
                            .expects('reportCallback')
                            .never();

  stockfetch.printReport();
  callbackMock.verify();
});
```

The first test confirms that the callback was called with the right data while the second test confirms that the callback was not called if all symbols have not been processed. Implement the minimum code to make these two tests pass before continuing to the next step.

Design sortData

The last two tests in the test list say that the printReport function should sort prices and errors. On second thought, implementing that code in printReport may lead to duplication. Let's revisit these tests:

- ...
- ✓ printReport does not send before all responses arrive
- ~~printReport sorts prices based on the tickers~~
- ~~printReport sorts errors based on the tickers~~
- printReport calls sortData, once for prices, once for errors
- sortData sorts the data based on the symbols

Time to roll out the last few tests:

testnode/stockfetch/test/stockfetch-test.js

```
it('printReport should call sortData once for prices, once for errors',
  function() {
  stockfetch.prices = { 'GOOG': 12.34 };
  stockfetch.errors = { 'AAPL': 'error' };
  stockfetch.tickersCount = 2;

  var mock = sandbox.mock(stockfetch);
  mock.expects('sortData').withArgs(stockfetch.prices);
  mock.expects('sortData').withArgs(stockfetch.errors);

  stockfetch.printReport();
  mock.verify();
});

it('sortData should sort the data based on the symbols', function() {
  var dataToSort = {
    'GOOG': 1.2,
    'AAPL': 2.1
  };

  var result = stockfetch.sortData(dataToSort);
  expect(result).to.be.eql([['AAPL', 2.1], ['GOOG', 1.2]]);
});
```

In those two short tests we verified first, that printReport calls sortData twice, and second, that sortData does its business as expected. The implementation should not take much effort; let's take care of that next.

```
this.printReport = function() {
  if(this.tickersCount ===
```

```
    Object.keys(this.prices).length + Object.keys(this.errors).length)
  this.reportCallback(this.sortData(this.prices), this.sortData(this.errors));
};

this.sortData = function(dataToSort) {
  var toArray = function(key) { return [key, dataToSort[key]]; };
  return Object.keys(dataToSort).sort().map(toArray);
};

this.reportCallback = function() {};
```

All the tests in the test list are now complete. We started with the readTickersFile function and walked through all the functions needed up through the printReport function. The tests are passing, but we can't call this done until we run the program and see the result on the console.

Integrate and Run

We need one final function that integrates all the functions together and for that, we need a few integration tests. Let's start with the integration tests; you can also key these into the stockfetch-test.js file:

```
it('getPriceForTickers should report error for invalid file',
  function(done) {
  var onError = function(error) {
    expect(error).to.be.eql('Error reading file: InvalidFile');
    done();
  };
  var display = function() {};

  stockfetch.getPriceForTickers('InvalidFile', display, onError);
});

it('getPriceForTickers should respond well for a valid file',
  function(done) {
  var onError = sandbox.mock().never();

  var display = function(prices, errors) {
    expect(prices.length).to.be.eql(4);
    expect(errors.length).to.be.eql(1);
    onError.verify();
    done();
  };

  this.timeout(10000);

  stockfetch.getPriceForTickers('mixedTickers.txt', display, onError);
});
```

The second test reads a file named mixedTickers.txt. Since this is an integration test, we need this file—we don't have the option of mocking file access this

time. You can readily use the file provided in your workspace, with the following content:

```
GOOG
AAPL
INVALID
ORCL
MSFT
```

The new tests exercise an integrating function getPriceForTickers, which takes three arguments: a filename, a callback to display prices and errors, and finally a callback to received errors related to file access. The first test checks to make sure the integrating function wires the error handler properly to the rest of the code. The second test ensures that it wires the display callback function properly. Add the following short integrating function to the StockFetch class in the stockfetch.js file:

```
this.getPriceForTickers = function(fileName, displayFn, errorFn) {
  this.reportCallback = displayFn;
  this.readTickersFile(fileName, errorFn);
};
```

That was quite easy. Run the tests and ensure all the tests are passing:

```
Stockfetch tests
  ✓ should pass this canary test
  ✓ read should invoke error handler for invalid file
  ✓ read should invoke processTickers for valid file
  ✓ read should return error if given file is empty
  ✓ parseTickers should return tickers
  ✓ parseTickers should return empty array for empty content
  ✓ parseTickers should return empty array for white-space
  ✓ parseTickers should ignore unexpected format in content
  ✓ processTickers should call getPrice for each ticker symbol
  ✓ processTickers should save tickers count
  ✓ getPrice should call get on http with valid URL
  ✓ getPrice should send a response handler to get
  ✓ getPrice should register handler for failure to reach host
  ✓ processResponse should call parsePrice with valid data
  ✓ processResponse should call processError if response failed
  ✓ processResponse should call processError only if response failed
  ✓ processHttpError should call processError with error details
  ✓ parsePrice should update prices
  ✓ parsePrice should call printReport
  ✓ processError should update errors
  ✓ processError should call printReport
  ✓ printReport should send price, errors once all responses arrive
  ✓ printReport should not send before all responses arrive
  ✓ printReport should call sortData once for prices, once for errors
  ✓ sortData should sort the data based on the symbols
```

```
  ✓ getPriceForTickers should report error for invalid file
  ✓ getPriceForTickers should respond well for a valid file (1181ms)

27 passing (1s)
```

That was quite a journey. It's great to see all the tests pass, but we also want to see the program run and print the results. Let's write a driver program to run the code. Open the file src/stockfetch-driver.js in the current project in the workspace and enter the following code:

testnode/stockfetch/src/stockfetch-driver.js

```
var Stockfetch = require('./stockfetch');

var onError = function(error) { console.log(error); };

var display = function(prices, errors) {
  var print = function(data) { console.log(data[0] + '\t' + data[1]); };

  console.log("Prices for ticker symbols:");
  prices.forEach(print);

  console.log("Ticker symbols with error:");
  errors.forEach(print);
};

new Stockfetch().getPriceForTickers('mixedTickers.txt', display, onError);
```

The code is not much different from the last integration tests we wrote, except we log the output to the console instead of programmatically asserting the response. To run the driver, use the following command:

```
node src/stockfetch-driver.js
```

Take a look at the output—of course, the data will be different when you run it, depending on the current market and network connectivity:

```
Prices for ticker symbols:
AAPL    105.260002
GOOG    758.880005
MSFT    55.48
ORCL    36.529999
Ticker symbols with error:
INVALID 404
```

We took a test-driven approach to design this application—it's great to see the fruits of that effort in action. Let's next take a look at the code coverage.

Review the Coverage and Design

A total of twenty-seven tests came together to influence the design of the code. Since we wrote the tests first, the behavior of every single line of code is verified by one or more tests. We'll confirm that's true by taking a peek at the code coverage and then discuss how the design has evolved.

Measure Code Coverage

Each time we ran npm test, the command has been quietly measuring the coverage, thanks to Istanbul. If you'd like to know the exact command to create the coverage report, take a look at the test command, under scripts, in the package.json file. You'll find the following command in there:

```
istanbul cover node_modules/mocha/bin/_mocha
```

Take a look at the coverage report produced as part of running the npm test command:

```
  Stockfetch tests
...
  27 passing (1s)

============================= Coverage summary =============================
Statements   : 100% ( 59/59 )
Branches     : 100% ( 10/10 )
Functions    : 100% ( 19/19 )
Lines        : 100% ( 55/55 )
============================================================================
```

If you'd like to see the line-by-line coverage report, open the file coverage/lcov-report/index.html in a browser.

In contrast to 307 lines in stockfetch-test.js, the source code is 96 lines in stockfetch.js. That's a 3-1 test/code ratio—quite typical. For example, in an e-commerce system in production at my organization, the ratio is 3.9-1.

Test/Code Ratio

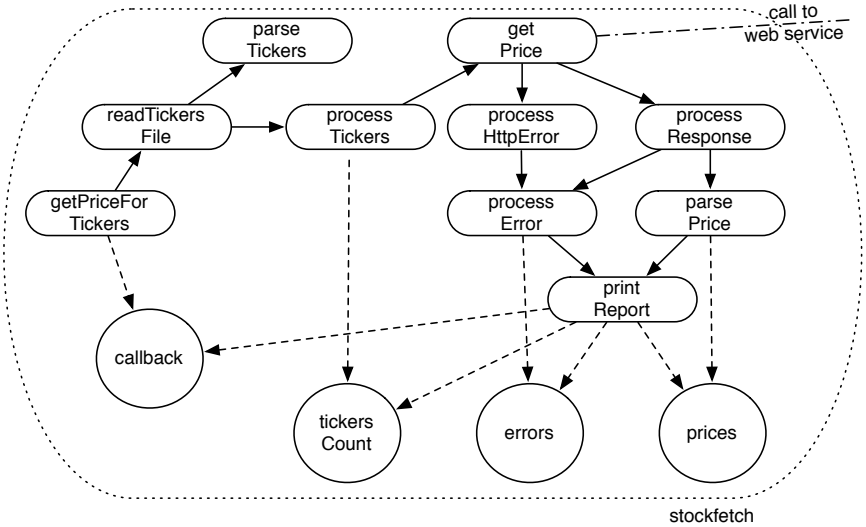
Applications that are test driven typically have three to five lines of test for each line of code.

Excluding the integration test, all the tests run pretty fast, are deterministic, and don't need a network connection to provide quick feedback that the code behaves as expected. While the two integration tests rely on a network connection, they too are deterministic in the way they're written. Next, we'll discuss the design that resulted from the test-driven effort.

The Design in Code

The tests and code evolved incrementally and it's hard to see the overall design from short code snippets. Download the electronic version of the test file[1] and the source code.[2] Take a few minutes to study both.

We started with a strategic design—see the figure in *Start with a Strategic—Just Enough—Design, on page ?*. That design gave us an initial direction, and the tests shaped that initial idea into a practical design with several good qualities. Each function is modular and concise, and does one thing. Along the way we applied many design principles—single responsibility principle (SRP), Don't Repeat Yourself (DRY), high cohesion, low coupling, modularity, separation of concerns, dependency injection… Let's visualize the resulting design.



1. https://media.pragprog.com/titles/vsjavas/code/testnode/stockfetch/test/stockfetch-test.js
2. https://media.pragprog.com/titles/vsjavas/code/testnode/stockfetch/src/stockfetch.js

Compare the tactical design diagram with the strategic design diagram. While the tactical design diagram has more details and modular functions, you can see the original functions we identified in the strategic design—readTickersFile, processTickers, getPrice, printReport—in their new enhanced forms. The interfaces of these methods evolved significantly during the tactical design. New supporting functions also cropped up during the design, mainly to make the code cohesive, modular, and above all testable.

It did take a lot more effort than it would to simply hack the code. However, we know the consequence of hacking code: it leads to unmaintainable code. When we invest the time to create better-quality code, it yields better dividends when the code changes to add features or fix bugs. We don't have to manually exercise every piece of code to know things behave as expected. Quick feedback equals greater confidence and agility.