What is the difference between a test runner, testing framwork, assertion library, and a testing plugin?

16 March 2015

4 Classifications of Testing Software

When first being introduced to the source code of real world testing suites it can be overwhelming how many levels of abstraction exist. The different levels of abstraction in testing suites can make it difficult to form an accurate mental representation of how things really work and what the role of each peice of testing software is. Each peice of software operates on a different level of abstraction and serves a unique purpose. My primary goal for this article is to offer some insight on the different levels of abstraction in testing stacks. I intend to shed some light on how these various levels of abstraction integrate together in a testing suite.

There are many different levels of abstraction in testing. It can be overwhelming to establish an accurate mental model of how real world testing suites are architectured. To resolve this cognitive dissonance I have created 4 major classifications that nearly all testing software fits into. The classifications I am providing are a good way to think about large testing suites in real world applications. However, this system of thinking is not the end all be all. As you explore testing much deeper, you may find that there are aspects of some testing suites that do not fall neatly into the classification schema I am providing. With that disclaimer aside, the 4 major classifications of testing software are:

1. Test Runners
2. Testing Frameworks
3. Assertion Libraries
4. Testing Plugins

Each classification essentially represents a different level of abstraction. I am going to spend this article delving into an example implementation of each classification. I will be demonstrating each classification by providing a concrete example of a peice of software that falls into that given classification. I will also demonstrate how these various pieces of software fit together.

- Karma (http://karma-runner.github.io/0.12/index.html) is an example of a **test runner**
- Mocha (http://mochajs.org/) is an example of a **testing framework**
- Chai (http://chaijs.com/) is an example of an **assertion library**
- Sinon (http://sinonjs.org/) is an example of a **testing plugin**

Karma: Test Runner

Karma is a type of test runner which creates a fake server, and then spins up tests in various browsers using data derived from that fake server. Karma is only a test runner, and requires a testing framework such as Mocha to plug into it in order to actually run tests.

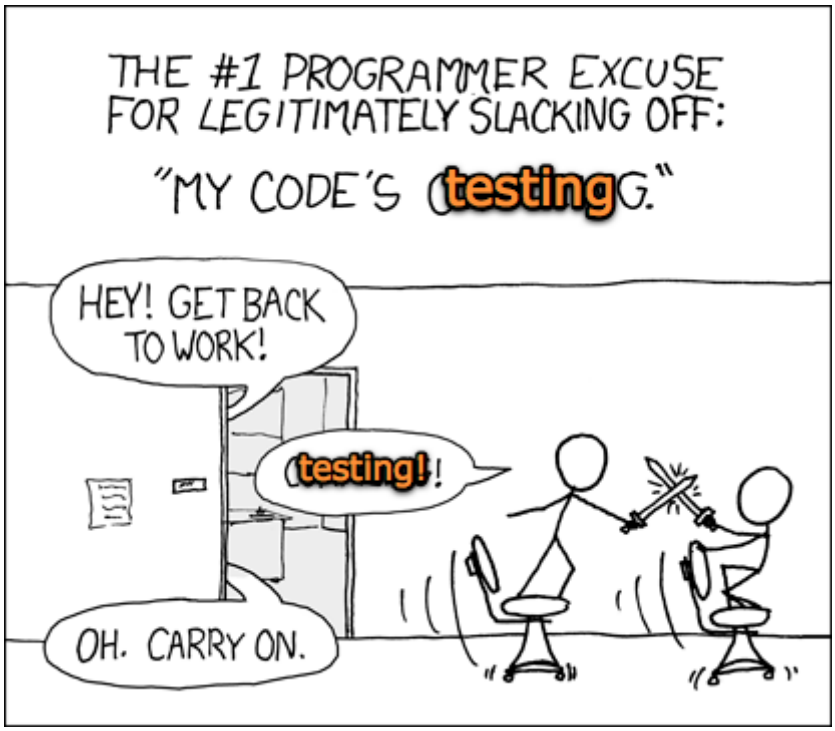
Test Runners work on the highest level of abstraction out of all testing software. All the other testing software takes place within the test runner. As a result, it is necessary to configure the test runner to work with the other testing software which plug into it. In our example of using Karma as a test runner, **`karma init`** is the command which creates the **`karma.conf.js`** file. This file is where all our test runner configurations will be.

The header of each karma configuration file follows this structure:

```
module.exports = function(config) {
  config.set({
    basePath: '../..',
    frameworks: ['mocha']
```

As you can see, this configuration file offers us a hook to include our testing framework, which exist on a lower level of abstraction than our test runner.

Additionally, Karma uses other technologies such as the minimatch library (https://github.com/isaacs/minimatch), which uses regex-like syntax to filter the correct files to use. The **`basePath`** property determines the root path Karma uses to find the actual specs, which exist on a lower level of abstraction.



Mocha: Testing Framework

The following file uses Mocha as a testing framework, and Chai as an assertion library:

```
describe('the todo.App', function() {
  context('the todo object', function(){

    it('should have all the necessary methods', function(){
      var msg = "method should exist";
      expect(todo.util.trimTodoName, msg).to.exist;
      expect(todo.util.isValidTodoName, msg).to.exist;
      expect(todo.util.getUniqueId, msg).to.exist;
    });
  });
});
```

How to Distinguish Between Mocha and Chai

How can we distinguish between which level of abstraction each part of the testing suite is occurring on? In other words, we are now working on a lower level than a test runner, we are actually writing tests now. We are using a lot of methods which are native to JavaScript. How do we distinguish which method is from which piece of testing software? It was easy to differentiate between a test runner and a framework. But now we are throwing a framework and an assertion library all in the same file. It is becoming muddled as to which method is a result of which piece of software.

We can distinguish between framework (Mocha) methods and assertion library (Chai) methods by looking at the contents of the **`it`** block. Methods outside the **`it`** block are generally derived from the testing framework. Everything within the **`it`** block is code coming from the assertion library. **`beforeEach`** , **`describe`** , **`context`** , **`it`** , are all methods extending from Mocha. **`expect`** , **`equal`** , and **`exist`** , are all methods extending from Chai.

```
afterEach(function() {
    $httpBackend.verifyNoOutstandingExpectation();
    $httpBackend.verifyNoOutstandingRequest();
    $window.localStorage.removeItem('com.shortly');
  });

  it('should have a signup method', function() {
    expect($scope.signup).to.be.a('function');
  });
```

All the methods concerned with the testing framework are occurring outside the **`it`** block, and all methods concerned with the assertion library are occurring inside the it block. Therefore we can conclude that anything occurring inside the it block is indeed occurring on a lower level of abstraction than the testing framework. Or in terms of our classification schema, everything occurring inside the it blocks is either part of an assertion library or a part of a testing plugin. The notion that anything inside the it block is occurring on a lower level of abstraction than the testing framework is only a heuristic, that is- it is merely a rule of thumb.

There are many technologies out there and unlimited edge cases to predict, but as a generality it would be fair to view the it block as the interface between two different levels of abstraction in testing.

Chai: Assertion Library

Chai is an assertion library that plugs into Mocha. Up until now we have just been concerned with using a test runner for automation, and using a testing framework for setup, teardown, and structure. But now we are getting into the meat and potatoes. The assertion library is what actually runs the specs and determines whether any given condition is valid or not. Ultimately, every test is ran by methods which are derived from our assertion library. It is worth mentioning though, not every framework needs an external assertion library. Jasmine (http://jasmine.github.io/) for example, has it's own assertion library builtin. Mocha is just structured in such a way where it **does** need an external assertion library. This makes Mocha more difficult to setup initially, but offers much greater flexibility than frameworks which use a builtin assertion library such as Jasmine.

Sinon: Testing Plugin

Sinon is a plugin which hooks into Chai and gives us the ability to perform a more diverse set of tests. Through the Sinon plugin we can create mocks, stubs, and fake servers:

```
describe('API integration', function(){
  var server, setupStub, JSONresponse;

  beforeEach(function() {
    setupStub = sinon.stub(todo, 'setup');
    server = sinon.fakeServer.create();
  });

  it('todo.setup receives an array of todos when todo.init is called', function () {
  });

  afterEach(function() {
    server.restore();
    setupStub.restore();
  });
});
```

Sinon has a bunch of cool features that allow you to really get into the nooks and crannies of your source code and see what is really going on under the hood.

Conclusion

I hope this article helped you by providing a good mental model in which to view large scale testing suites. Of course no heuristic is a substitute for playing with it on you're own and getting a feel for it. Once you start writing specs using all these different software components you will get an intuition of how they all fit together. As we are exposed to different systems, over time we become more adept at noticing underlying patterns which determine how those systems operate. My purpose for writing this article was to uncover some underlying design patterns in testing systems. Hopefully this article will make you more cognizant of some of

these underlying patterns. Pattern recognition is only the first step though, after pattern recognition you need to move onto pattern utilization, and not long after that you will by default move into the ultimate stage of engineering: pattern creation. I hope you found this article interesting and insightful. Have a good one!

testing 1 (/categories.html#testing-ref)

javascript 11 (/tags.html#javascript-ref)

software architecture 4 (/tags.html#software architecture-ref)

programming paradigms 3 (/tags.html#programming paradigms-ref)

design patterns 5 (/tags.html#design patterns-ref) testing 1 (/tags.html#testing-ref)

« Previous (/technology%20feature/2015/03/10/technology-feature-tweettrackerjs)

Archive (/archive.html)

Next » (/computer%20science/2015/03/20/representing-decision-trees-with-code)
