LOGIC ROOM

Test Driven Development Simplified in 5 Steps

by Pete Heard

Table of Contents

Introduction & How to Use This Guide

For a long time, Test Driven Development (TDD) baffled me. I followed all the guidelines and rules that are commonplace when you begin to practice it. I read lots about it, tried it out on various different projects and discussed it with colleagues. Many of the people I spoke to were dismissive of it; they thought it was overly complex and dogmatic. And try as I might I had to agree at least in some part because I was struggling myself to really believe it was a useful discipline.

One summer I decided to start my own project in the evening where I could have total freedom to experiment. At around the same time I decided to soak up all the major online resources relating to TDD. To keep an open mind, I decided I would look for the good and bad of TDD. What happened was quite surprising. By focusing on the negative and the positives I found I was able to do what I had not been able to do before; 'filter' out the bad advice. Or to be more specific, find a subset of advice that wasn't contradictory!

Since then I have been on numerous projects using TDD and have seen it implemented in different ways with varying success. I wrote this guide as an extension of a talk I do called 'TDD Simplified'. It is a simple guide to try and help you avoid the pitfalls I have both made and seen. It should be used as a baseline to apply TDD. It is designed for anyone who is feeling that they are not getting high value from their approach or who has not started yet and wants to avoid making the same rookie mistakes as I did!

There are 5 basic steps to simplifying TDD.

1) Observing Objectives
2) Layering
3) Decoupling Tests from Implementation Details
4) Extracting the benefits of BDD and Integration Testing
5) Domain and Framework Separation

It is by no means the final word on TDD. But these are five areas that I have come to consider important starting points to be able to really get value whilst minimizing overhead.

The e-book uses AngularJS as a Single Page Application (SPA) framework and the JavaScript language. If you are reading this and your experience is with strongly typed languages just be aware that JavaScript is dynamic.

Hopefully the things I am sharing with you will help you experience what I do from TDD, value and enjoyment!

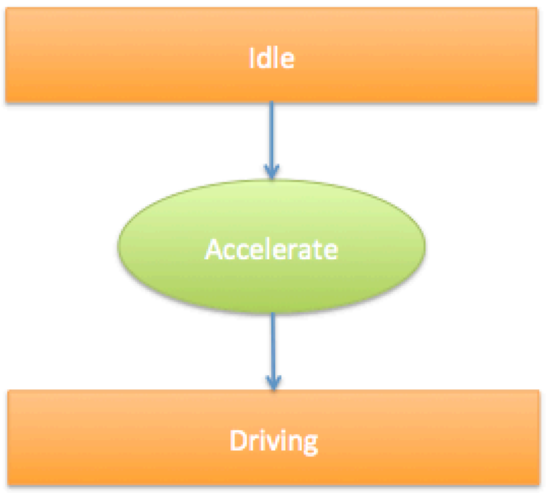
Foundation Knowledge

The State Machine, Classic TDD vs Mockism

There are two schools of thought when it comes to TDD. The first is the 'Mockist'. The Mockist believes in isolating every subsystem within a piece of software and testing it in isolation. They are known by the colloquial term 'Mockists'. You can read more here. My success with TDD (Like Martin Fowlers) has come through taking the other approach; the 'Classic' approach. This guide uses the classic approach.

In the classic approach of TDD it is applied in terms of the simple 'state machine'. A state machine has 2 states and 1 operation. For example; if I were to describe my car as a simple state machine I would say it had 2 states; (idle and driving), and 1 operation (accelerate). If I were to draw a visual representation of my car as a state machine it would look something like this....

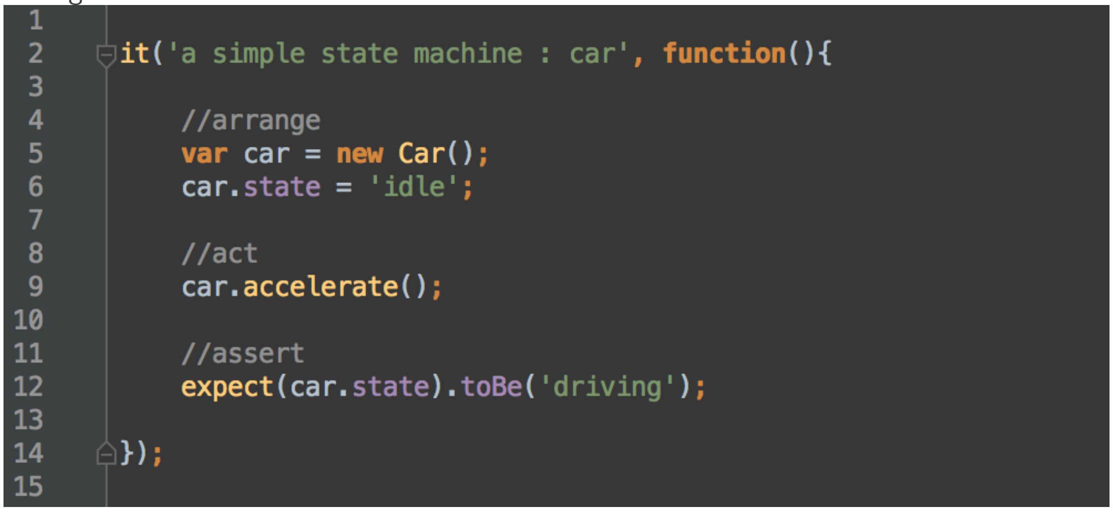
Fig 0-1



Thinking about each test as a state machine is useful to be able to accurately and confidently pinpoint exactly what you are testing and how you expect it to behave.

Many TDDists like to use the 3 verbs (Arrange, Act, Assert), which describe a state based approach to testing. In fact, it is quite common to see this in a unit test. For example, the following unit test is what I might see if someone were testing the car from my example.

Listing 0-2



```

it('a simple state machine : car', function(){

    //arrange
    var car = new Car();
    car.state = 'idle';

    //act
    car.accelerate();

    //assert
    expect(car.state).toBe('driving');

});

```

We can see here a very clear manifestation of the state based machine in code. We set our car up in its 'idle' state. We accelerate. Then we check that it is in the 'driving' state afterwards.

NOTE: It is very hard to do TDD with code that does not have a fixed known and end state. So be mindful when testing to know that you are dealing with a fixed system. For example, one common problem is dealing with clocks or dynamic ID generation. When dealing in a state based system try to lock these variables down so that you can create more stable tests!

Stubbing and Spying On IO Boundaries

When taking a state based approach to testing we will invariably want to intercept code at the point that data comes into the system or calls are made out from the system. Why? Because these are the boundaries of our application. They tell us about the interface through which our application will deal with the real world. The way we do this is with two types of Test Double**, they are…

Stubs:

If I am writing a booking system, I would want to be able to exercise my code in a known state. This means loading some sort of data record into the code forcefully when I run the test so that I control how the data looks and thus the program behaves. I need to do this in order to test the state. So, in this scenario I would load a test double known as a 'Stub'.

Spies:

If I wanted to check that my production code sends the correct data to some other location using some sort of IO operation, I need to use a Spy. A Spy will allow me to 'peek' at the very last line of code before my program executes but before it actually performs the IO operation. This way I can know what I am sending before I send it. So, in this scenario I would load a test double known as a 'Spy.

Both stubs and spies will allow us to exercise our test code WITHOUT performing IO operations. Being able to do this is a fundamental pre-requisite of writing high quality and fast execution unit tests. To learn more about all types of test doubles check out Uncle Bobs blog 'The Little Mocker'

Now let's move onto the 5 steps...

Observing Objectives (80/20 rule)

The world of software development is full of new stuff to learn. Learning in software is the thing you do most. If you aren't learning a new framework you are learning how someone's code works. If you aren't learning how someone's code works you are learning new skills or soaking up new knowledge from books, I could go on… learning takes time. TDD is no different. It is a skill to be learned and this means you need to invest time to make it work. In order to keep your investment of time effective this section exists to help you answer two fundamental questions:

- What is TDD?
- What are the objectives?

What is TDD?

TDD stands for Test Driven Development. The general idea is that you will:

1) Write a unit test that will test implementation that doesn't yet exist (test fails)
2) Write code that will allow the unit test to complete (test works)
3) Refactor the underlying code and unit test until they are clean and well designed

As you can see TDD is just a workflow for creating tests, that's all. It is a simple lightweight process that will result in you having automated testing to go with your code.

What are the objectives?

In order to gain insight into what you are trying to achieve take a step back for one moment and consider some important objectives with regards to any automated testing regardless of the process you use…

- The automated testing should cover as much functionality as necessary (it should give you confidence that your code works)
- The automated testing should be fast (since shorter feedbacks in development are cheaper to resolve)
- The automated tests are readable and maintainable (so they stand the test of time)
- The automated tests are written first or at the same time (so that you don't forget to test something)

- The code and automated tests are easy to refactor (refactoring is difficult but extremely important, so best not to place any barriers in the way)

When a project begins everyone starts with the honorable intention to create 'better software'. Roughly translated this means they will want to improve on what they didn't do well last time. Usually before a project kicks off everyone sits down to decide which approaches will be used to deliver the project successfully. When you go through this process if you decide you want to use TDD keep in mind the above objectives in order to guide your decision making process on the best way to implement your automated testing using TDD.

By being objective about your decision making you will be able to get the maximum amount of value from the minimum effort that you will apply. I like to keep in mind the 80/20 rule when implementing TDD. I try to find the simplest way to achieve a large proportion of the benefits.

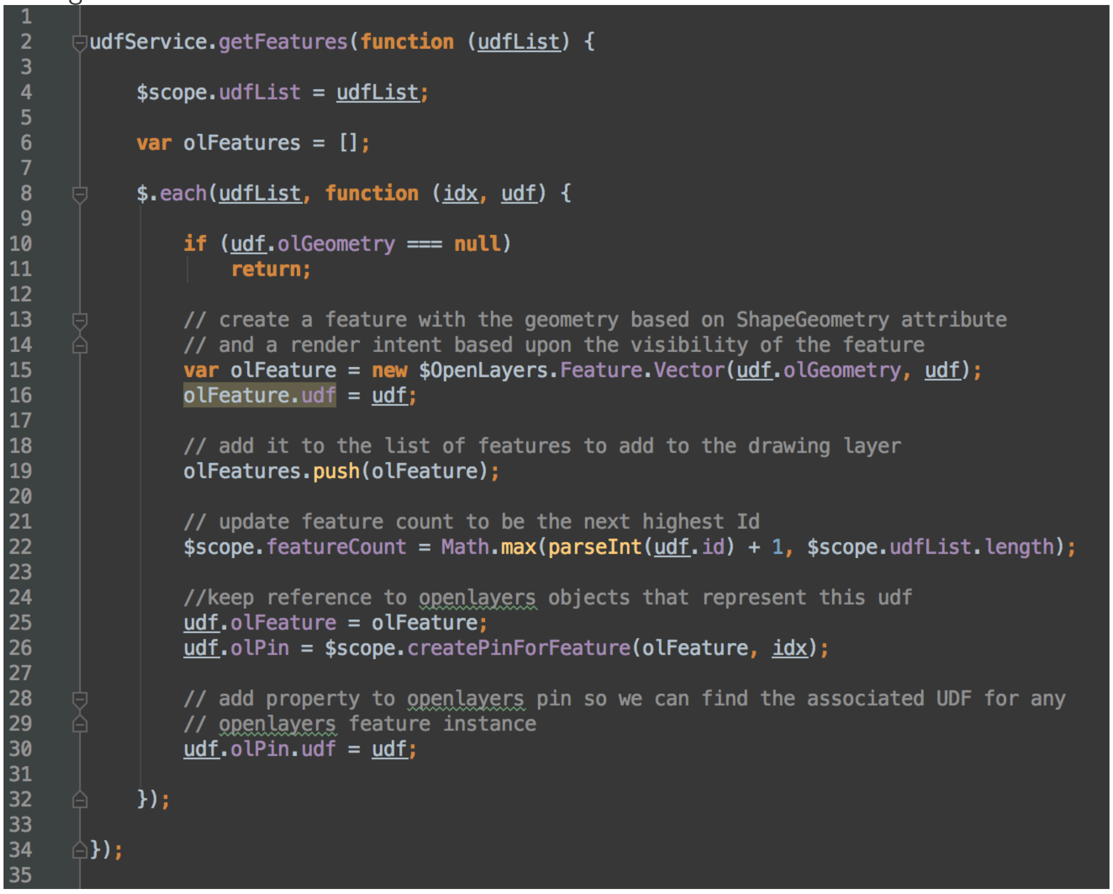
Bottom Line:

Keep your objectives in clear sight when implementing TDD. Remember test speed, coverage and readability are key objectives for a sound automated test strategy.

Layering

I was taking a course with Pluralsight a few years ago and a very interesting term was coined 'Cognitive Load'. The trainer, to his credit wanted to deliver the course to us in a way that would reduce the cognitive load on our minds. This meant slowly loading us up with information in successive steps as opposed to giving it all to us in one entire go. Consider the following code fragment of a mapping component....

Listing 2-1



```
1
2  udfService.getFeatures(function (udfList) {
3
4      $scope.udfList = udfList;
5
6      var olFeatures = [];
7
8      $.each(udfList, function (idx, udf) {
9
10         if (udf.olGeometry === null)
11             return;
12
13         // create a feature with the geometry based on ShapeGeometry attribute
14         // and a render intent based upon the visibility of the feature
15         var olFeature = new $OpenLayers.Feature.Vector(udf.olGeometry, udf);
16         olFeature.udf = udf;
17
18         // add it to the list of features to add to the drawing layer
19         olFeatures.push(olFeature);
20
21         // update feature count to be the next highest Id
22         $scope.featureCount = Math.max(parseInt(udf.id) + 1, $scope.udfList.length);
23
24         //keep reference to openlayers objects that represent this udf
25         udf.olFeature = olFeature;
26         udf.olPin = $scope.createPinForFeature(olFeature, idx);
27
28         // add property to openlayers pin so we can find the associated UDF for any
29         // openlayers feature instance
30         udf.olPin.udf = udf;
31
32     });
33
34 });
35
```

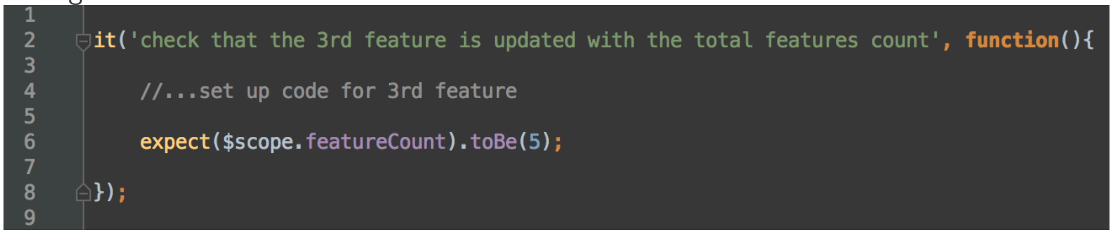
We can see that the (albeit ugly) code performs the following steps...

1. [line 2] Accesses a service that returns (in this case) a Data Transfer Object (DTO) called 'UdfList'.

2. [line 10] Makes a decision based on a business rule (if it has geometry then the business wanted the feature to be drawn a certain way).
3. [line 22] It updates a featureCount (whose sole purpose is to display a little number once the map is rendered).

Let's say for example that our unit test wanted to check the number referenced in step 3 (the featureCount) we would end up with a test like this....

Listing 2-2



```
it('check that the 3rd feature is updated with the total features count', function(){

    //...set up code for 3rd feature

    expect($scope.featureCount).toBe(5);

});
```

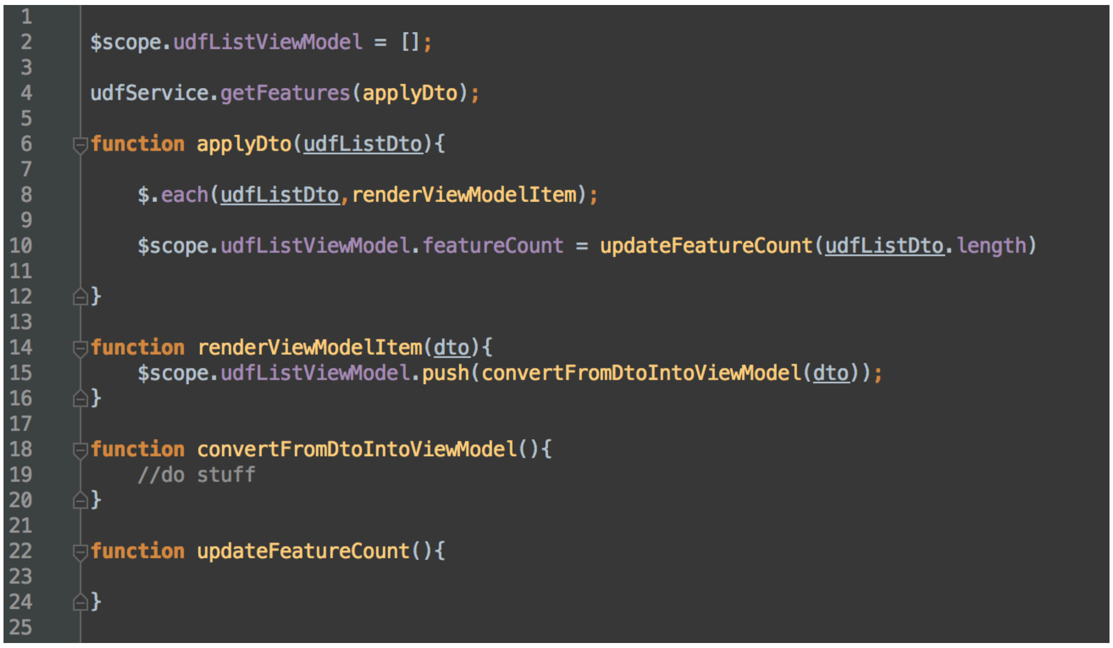
By looking at this test we cannot see what the feature count is for. We cannot be sure if it is something that will be:

1) Used in the view or...
2) Used internally in the application

Because the feature count is just sort of 'plonked' on the Angular scope (which is very similar in this context to using global scope) we get no knowledge of the architecture of the application, we get no idea about the intent and we have no idea about where this lives or why.

Consider the following refactoring...

Listing 2-3



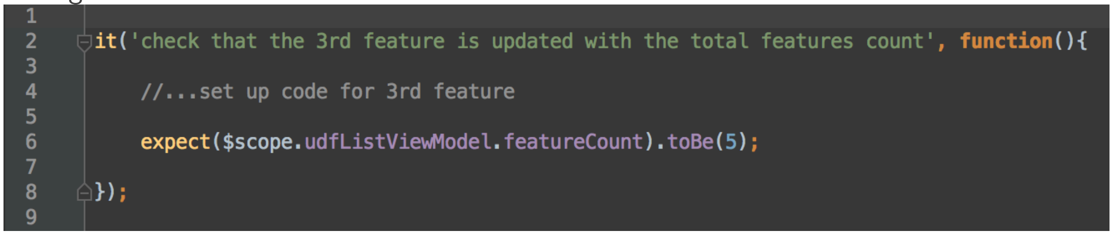
```
1
2    $scope.udfListViewModel = [];
3
4    udfService.getFeatures(applyDto);
5
6    function applyDto(udfListDto){
7
8        $.each(udfListDto,renderViewModelItem);
9
10       $scope.udfListViewModel.featureCount = updateFeatureCount(udfListDto.length)
11
12   }
13
14   function renderViewModelItem(dto){
15       $scope.udfListViewModel.push(convertFromDtoIntoViewModel(dto));
16   }
17
18   function convertFromDtoIntoViewModel(){
19       //do stuff
20   }
21
22   function updateFeatureCount(){
23
24   }
25
```

What we have attempted to do here is compose what the original code is doing in a number of discreet steps (which each function encapsulates). Notably; we have been very explicit about what is going to be displayed in the view by calling it ViewModel and assigning the value to it (line 10). There should be no doubt in anyone's mind that the ViewModel's object purpose is to be rendered by the view.

We have begun implementing the policy of '<u>Separation of Concerns</u>
Cognitive Load

Apart from there being a lower cognitive load in reading code which is separated into layers we can now write a test which ensures we are testing the outermost portion of our application (the ViewModel), we will be testing the end result, which is easier to read and reason with, it is also less tightly coupled; making refactoring easier. Our test can now be written like this...

Listing 2-4



```

it('check that the 3rd feature is updated with the total features count', function(){

    //...set up code for 3rd feature

    expect($scope.udfListViewModel.featureCount).toBe(5);

});

```

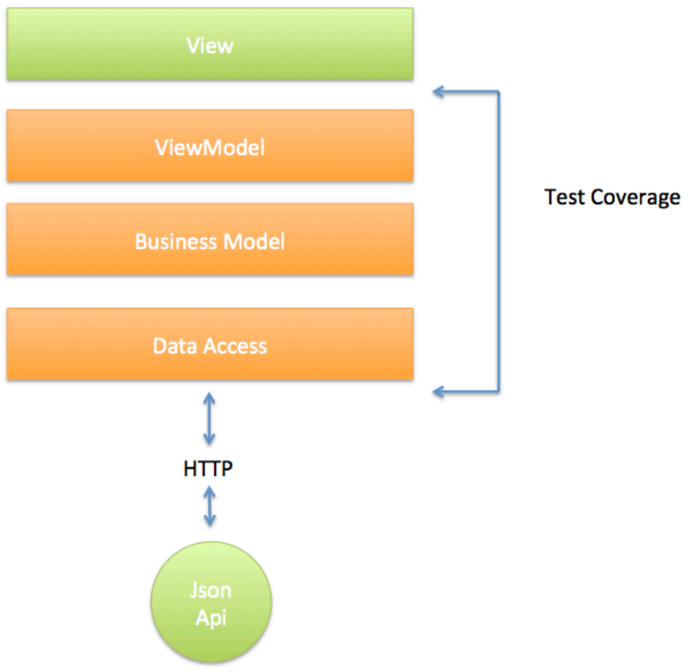
Now this test reads very easy. We can see that the featureCount is on the View Model.

Typical Layering

Quite often this layered approach will end up in 3 - 4 main conceptual layers. The diagram below (2-5) shows some layers of a common application. It also shows the coverage of the unit tests. Notice how deep they are compared to lower level tests.

When writing unit tests think about layers and their associated inputs and outputs. A well-layered system will define its inputs and outputs clearly so they can be tested easily.

Diagram 2-5



Bottom Line:

Layer your application to ensure that your tests can be focused on the correct area. This will also make code easier to read, refactor and comprehend ensuring maximum scalability for non-trivial projects.

Decoupling Tests from Implementation Details

In his book (Test Driven Development by Example) Kent Beck talks about his first insight into a 'test first' mindset. He speaks about when his father taught him to program and that he would know what needed to be outputted on the screen. He would sit at the computer and manipulate the program until the actual output matched the expected output.

The insight gained from that story is simple but powerful, that is: as a programmer we are trying to get the system output to match what we had in mind. Thus we have the concept of 'testing'. When you think about it testing is simply checking that a system meets some sort of defined specification.

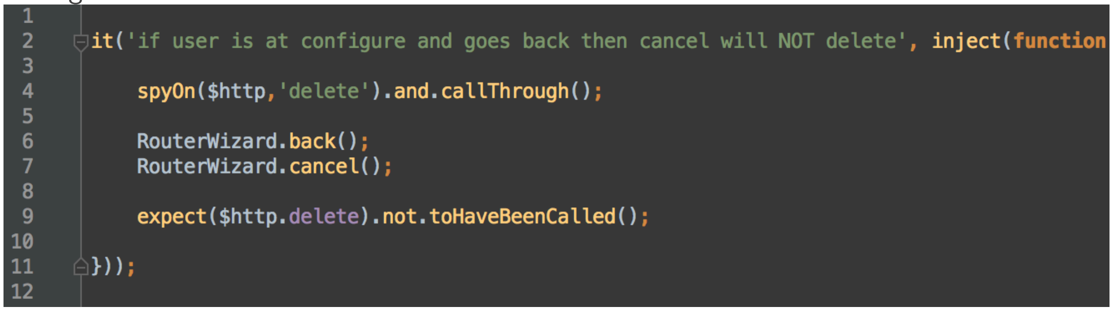
If TDD is a discipline used for the creation of automated tests, why should the way the unit tests work be different to how a manual test works?

Many unit tests are written at a very low-level, for example to test individual functions or classes/objects. It's important to realize the positive and negative of this approach. On the plus they can be fast to write and if they fail they tell you exactly where something has failed. On the downside because they are tightly coupled to the code you often need to understand the code to understand what the test is doing. Added to this, they make refactoring more difficult because not only do you have to refactor the code but you then have many tests to refactor. If your objective with TDD is to refactor often and clean your code, then it would be wise to test in a fashion that supports this objective.

What if we approached our automated testing in the same way as we approach manual testing? Our tests would test the behavior of the system not the underlying implementation and as a side effect we begin to couple our tests not to implementation but to public api functions.

Compare the following tests, the first (listing 3-1) tests behavior at a high level (i.e. it tests the 'Use Case' of the system). The second (listing 3 -2) tests low-level implementation.

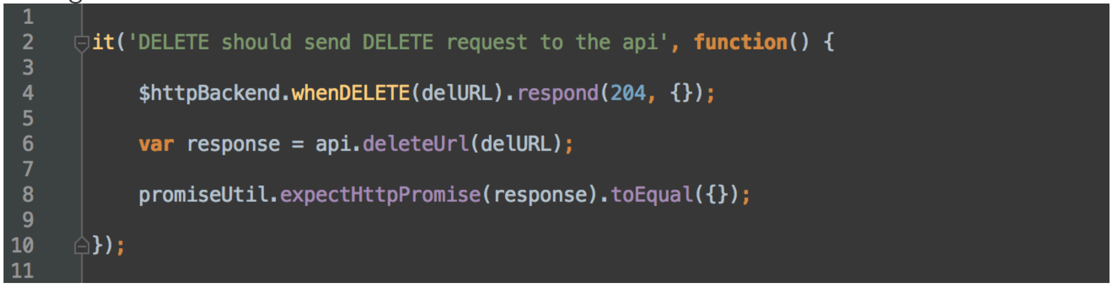
Listing 3-1



```
it('if user is at configure and goes back then cancel will NOT delete', inject(function

    spyOn($http,'delete').and.callThrough();

    RouterWizard.back();
    RouterWizard.cancel();

    expect($http.delete).not.toHaveBeenCalled();

}));
```

3-1 This test will invoke two public functions, which control a module, which is responsible for adding 'routers' (the RouterWizard). It was written to make sure that when the user goes 'back' that the cancel button will not delete the record on the server.

Listing 3-2



```
it('DELETE should send DELETE request to the api', function() {

    $httpBackend.whenDELETE(delURL).respond(204, {});

    var response = api.deleteUrl(delURL);

    promiseUtil.expectHttpPromise(response).toEqual({});

});
```

3-2 This second test will simply check a low-level API function to make sure that when a delete command is received that the response from the API is a JavaScript promise (which is just an object which has the result of the call).

Whilst both are valid tests, it's important to consider the benefits and drawbacks.

The low-level test at listing 3-2 will only test the unit in question. And the person reading it will have their mind pulled into reading information about low-level implementation details such as HTTP responses. This means they have to understand and remember HTTP specific details (such as response codes). They will also need to understand JavaScript promises for it to make any sense.

The high-level test at listing 3-1 may be able to test both some functionality AND some behavior at the low-level, for example notice the Spy at (line 9) which is Spying on the http endpoint to make sure it was called. This high-level test is only coupled to the public API of the code or the final IO boundary* (the $httpBackend) meaning that when refactoring internal code, we are less likely to need to juggle the test code as much. One of the fundamental principles behind OO programming is that of 'data hiding'. The higher test (listing 3-1) works better with this idea since it promotes hiding the internals of the application from the test code.

By focusing tests around the public API we begin to ignore internal implementation details. We start to see only a few high-level public functions need be tested and our unit testing can ignore that private and internal functions and properties.

*IO Boundary This is the last line of code before an IO operation happens. For example; calling framework code that creates a HTTP request is an IO operation. Or making the browser redirect is an IO operation.

Bottom Line:

Reduce test to code coupling and increase readability by considering testing the Use Case of the system first. This approach will result in tests focusing on the public API of your code and will help to create a clean public API. If the Use Case covers your objectives for the test, then ask yourself do you need low-level testing as well?

Extracting the benefits of BDD and Integration Testing

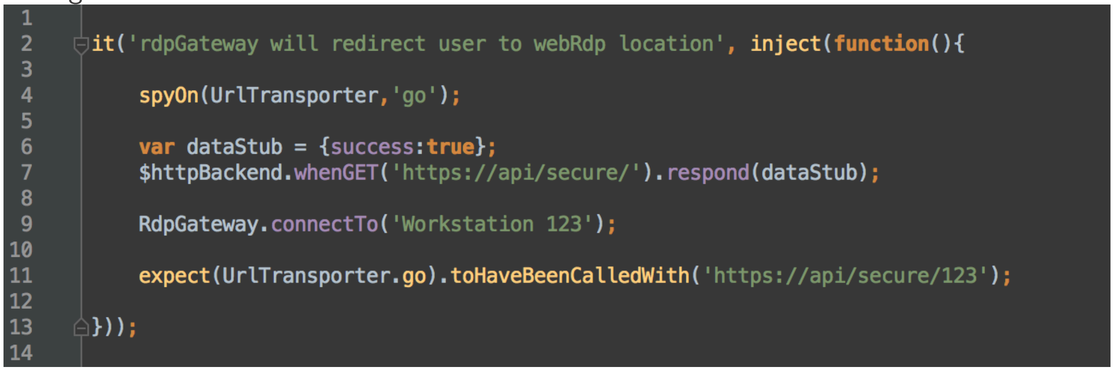
Quite often the related topics of BDD (Behavior Driven Development) and Integration Testing will be thrown into the mix when deciding a testing strategy, these are fascinating topics in their own right, let's have a quick look at what they offer over and above pure TDD.

- BDD provides a ubiquitous language, which can help with stakeholder buy-in and business language integration into the software development and testing workflow. This is built on top of unit testing.
- Integration Testing provides a very robust way to get a deep and thorough test of large portions of functionality.

So what if we could take the valuable concepts behind BDD and Integration Testing and embed that into our TDD mindset? Then we get the benefits without necessarily needing the entire overhead. On many of the projects I have worked on we have done just that.

Consider the following code.

Listing 4-1



```
1
2   it('rdpGateway will redirect user to webRdp location', inject(function(){
3
4       spyOn(UrlTransporter,'go');
5
6       var dataStub = {success:true};
7       $httpBackend.whenGET('https://api/secure/').respond(dataStub);
8
9       RdpGateway.connectTo('Workstation 123');
10
11      expect(UrlTransporter.go).toHaveBeenCalledWith('https://api/secure/123');
12
13  }));
14
```

This code tests a module (RdpGateway) that allows the calling code to pass a Workstation ID to it (called Workstation 123). When that happens it will call a server and download a link from it using HTTP. Then it will redirect the browser to that link.

- The RdpGateway is the module under test and it has many sub-modules beneath it. But it exposes a single public API method. This is what I have chosen to test. After all, it is what calling code will use. I have chosen to test the 'behavior' of the component. And because of this I can write the description of the test (line 2) in a way that is easily comprehended. I could hand it to any person involved with the project and they would understand its intent without needing programming knowledge. Therein lies a key feature of BDD (a ubiquitous language).

- I have created a Spy on an IO boundary called the UrlTransporter (line 4) this is a module, which will talk to the browser directly (so must be Spied upon to stop the browser and test runner from actually redirecting). In addition to this I have created a Stub at the point in the program that makes a HTTP GET call (line 6-7) which will return subbed data rather than make a real HTTP call. Because we are only Stubbing and Spying on IO boundaries the code will execute almost the same as it will in a 'live' scenario. We end up with robustness like an integration test but because no IO is performed, it will run fast.

In this example you can see how by writing our test in a way that tests the behaviors of our code, using behavioral language and by using Stubs and Spies to block actual IO from happening we are getting some benefits of BDD and Integration Tests just by writing normal unit tests. We have exercised the code in exactly the same way as it would be exercised in production but we have made sure that no IO takes place in order to keep our test running fast!

Bottom Line:

Implement a lightweight TDD process, in which each test behaves like a BDD test, covers as much of code as an integration test (using Stubs and Spies) but runs really fast.

Domain and Framework separation

"A framework is all of the assumptions without any of the code" – anon

A lot of frameworks (Ruby, AngularJS, PHP) were created because the developers needed the frameworks themselves or they decided they needed to solve a particular issue. Understanding the motives of why something exists is often a good way to assess how you will use it.

Knowing that frameworks often began life as a tool for something or someone else gives us a valuable insight, it also ties up with the quote at the beginning of this section. There is every chance that you may have designed the framework differently if you were to make it fit with your application.

When dealing with frameworks I try and keep this in mind. A framework is a tool created by developers to solve a certain problem. For example, AngularJS (which I have used in this e-book) is a tool, which in my mind is very good at…

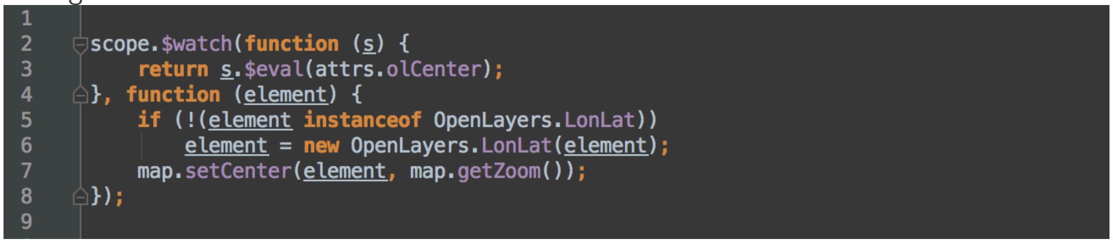
1) Providing a strong foundation for rendering views
2) Providing a mechanism for single page application behaviors
3) Dealing with http data

What a framework such as Angular is not good at is knowing about the application you are writing, of course that would be impossible. By understanding this we begin to see a framework as something to be used (but not entirely depended on for everything), it must be kept at arms distance because it is vital to keep the code you write separate from framework code if you want to reduce the overhead of testing.

Let's have a look at a quick code sample to see why…

Take the following code fragment, which uses two frameworks AngularJS and OpenLayers:

Listing 5-1



```
1
2  scope.$watch(function (s) {
3       return s.$eval(attrs.olCenter);
4  }, function (element) {
5       if (!(element instanceof OpenLayers.LonLat))
6           element = new OpenLayers.LonLat(element);
7       map.setCenter(element, map.getZoom());
8  });
9
```

The purpose of the function is to allow a user to hover over an element and when this happens a map will focus on it.

We can see that…

1) [line 2-3] The code is watching a function contained within a map for changes to the output of that function
2) [line 5-7] If the function watched (line 2-3) returns an OpenLayers geometry (OpenLayers.LonLat) then it will center the map on the last line (because there are elements on the map that are not LonLat types).

If we wanted to write a test that would assert that the map was centered when the user hovered over an item that can be centered on then we need to exercise the code. Because of this we would have two options.

1) We either include the original implementation of OpenLayers and use the actual framework to test the code.
2) Or we write Test Doubles* and overwrite the objects being tested meaning we do not have to use the framework.

Problems with the first approach

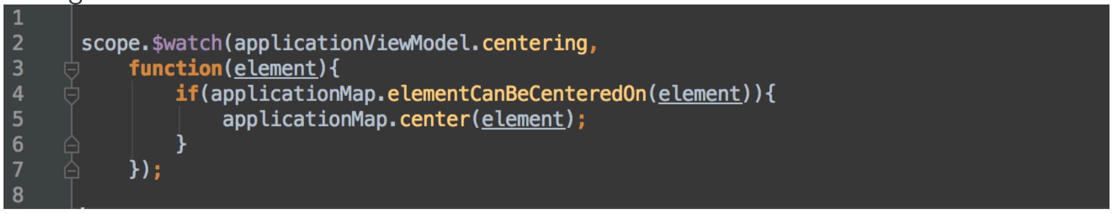
The problem with the first option is that simply including a reference to OpenLayers is not as simple as it sounds. Having worked with this framework on a number of occasions I know it will actually require quite a bit of set up (at least 5 lines). The chances of making a wrong assumption during this setup phase pose a risk. We will also have to make sure any objects passed into functions as the test runs are correctly set up (because they will be processed by OpenLayers) and if they aren't correct OpenLayers will throw an exception. And some of the objects are complex in themselves adding to the difficulty.

*Test Double This is a small portion of test code that is designed to replace portions of code within our test with functionality that will help the test. For this tutorial we focus on two types of double. The Mock and the Spy. Check out 'The Little Mocker' by Uncle Bob (in references below) for a great overview of test doubles.
Problems with the second approach

The problem with the second option is that if we use dynamic overwriting to create test double(s) they will have to implement the attrs.olCenter function on (line 3). Otherwise the test will crash, and then we will have to implement LongLat property on line (5-6) for the same reason. And finally, in order to check the final state, we will have to make sure that the map.setCenter() does something or sets something on (line 7) in order to be able to check that our final state is the required one.

Consider a refactored function:

Listing 5-2



```
scope.$watch(applicationViewModel.centering,
    function(element){
        if(applicationMap.elementCanBeCenteredOn(element)){
            applicationMap.center(element);
        }
    });
```

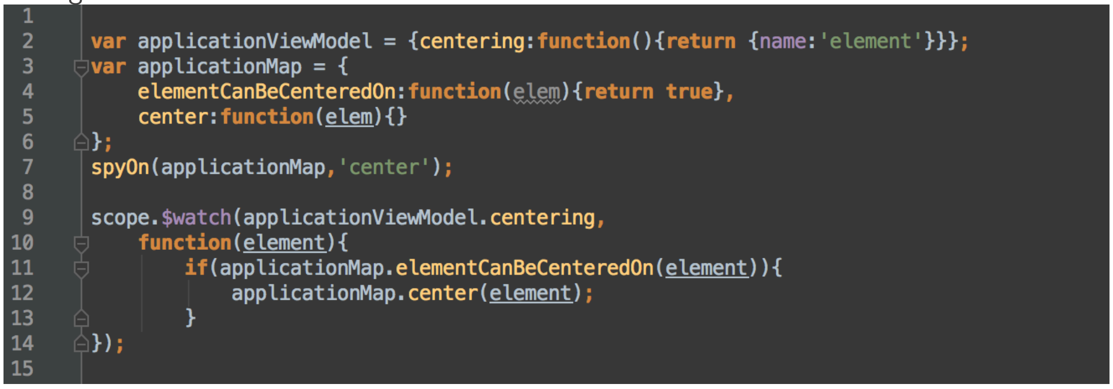
What we have done is introduce an interface (of our own choosing) to the program (we have Inverted Dependencies). The objects 'applicationViewModel' and 'applicationMap' are used as wrappers around the framework code. The framework code exists in another object which will be injected in using Dependency Injection. This means that we can more easily override the implementation of these during the test since the interface to them is much simpler.

By adding a layer of indirection between consuming and framework code it we have begun implementing the policy of 'Inversion Of Control'.

Solution

From a testing perspective we now don't have problems associated with either complex set up of the physical framework or the complexity of using dynamic type overriding of real types. We just create a very simple object that represents enough about the interface in order to check that the map is centered when the right pre-conditions of the test are met. The following code shows the full set up.

Listing 5-3



```
var applicationViewModel = {centering:function(){return {name:'element'}}};
var applicationMap = {
    elementCanBeCenteredOn:function(elem){return true},
    center:function(elem){}
};
spyOn(applicationMap,'center');

scope.$watch(applicationViewModel.centering,
    function(element){
        if(applicationMap.elementCanBeCenteredOn(element)){
            applicationMap.center(element);
        }
});
```

Apart from making testing easier, keeping a clear separation of your domain code and your framework code is just good practice, plain and simple! For one, if you ever need to change or update the framework it will be much easier since there is less coupling, and to someone reading your code you are saving them having to understand the internals of a framework. By introducing this interface

and decoupling from the framework we allow the code to read in a more orthogonal way and we hide the internal design of the framework from the calling code.

Another major benefit is the reduced amount of framework specific test code you will need; you will begin to test plain language code. This will help you to create tests fast especially at the start of a project where you are still learning 'how' to test your application.

Bottom Line:

Apart from making your code easier to understand and modify, keeping framework code and domain code separate is a great way to reduce the amount of effort you spend in creating unit tests that fill your test objectives. Using IOC (Inversion of Control) achieves this goal.

Recap

To sum up here are the 5 steps with their basic observations.

1) Observing Objectives

Keep your objectives in clear sight when implementing TDD. Remember test speed, coverage and readability are key objectives for a sound automated test strategy.

2) Layering

Layer your application to ensure that your tests can be focused on the correct area. This will also make code easier to read, refactor and comprehend ensuring maximum scalability for non-trivial projects.

3) Decoupling Tests from Implementation Details

Reduce test to code coupling and increase readability by considering testing the Use Case of the system first. This approach will result in tests focusing on the public API of your code and will help to create a clean public API. If the Use Case covers your objectives for the test, then ask yourself do you need low-level testing as well?

4) Extract the benefits of BDD and Integration Testing

Implement a lightweight TDD process, in which each test behaves like a BDD test, covers as much of code as an integration test (using Stubs and Spies) but runs really fast.

5) Domain and Framework Separation

Apart from making your code easier to understand and modify, keeping framework code and domain code separate is a great way to reduce the amount of effort you spend in creating unit tests that fill your test objectives. Using IOC (Inversion of Control) achieves this goal.

Finally

It's easy to get caught up in doing things perfectly with software. So when I start getting difficulty I like to peel back the layers until I can find a few simple things that work.

Hopefully this guide has demonstrated 'just enough' to help you either start or restart TDD. I am sure that if you can take just one of these steps and apply it your life with TDD will be more productive.

Resources

I highly recommend the following resources:

TDD where did it all go wrong (Ian Cooper)
https://vimeo.com/68375232

TDD is Dead (David Heinemeier Hanson)
http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html

Thoughtworks - TDD is dead debate (DHH + Kent Beck + Martin Fowler)
https://www.youtube.com/watch?v=z9quxZsLcfo

"By the way DHH has some valid points but his concerns can be avoided by following point 2 and 3 of this guide."

Video About TDD - Jason Goremen
https://www.youtube.com/watch?v=nt2KKUSSJsY

When TDD doesn't work – Uncle Bob
https://blog.8thlight.com/uncle-bob/2014/04/30/When-tdd-does-not-work.html

The Little Mocker - Uncle Bob
https://blog.8thlight.com/uncle-bob/2014/05/14/TheLittleMocker.html

About Pete Heard

Pete Heard is a full stack JavaScript developer who has spent over a decade learning to craft robust software using Test Driven Development and advanced Object Oriented design. He is the founder of Logic Room; a consultancy that helps organizations create web and mobile software.