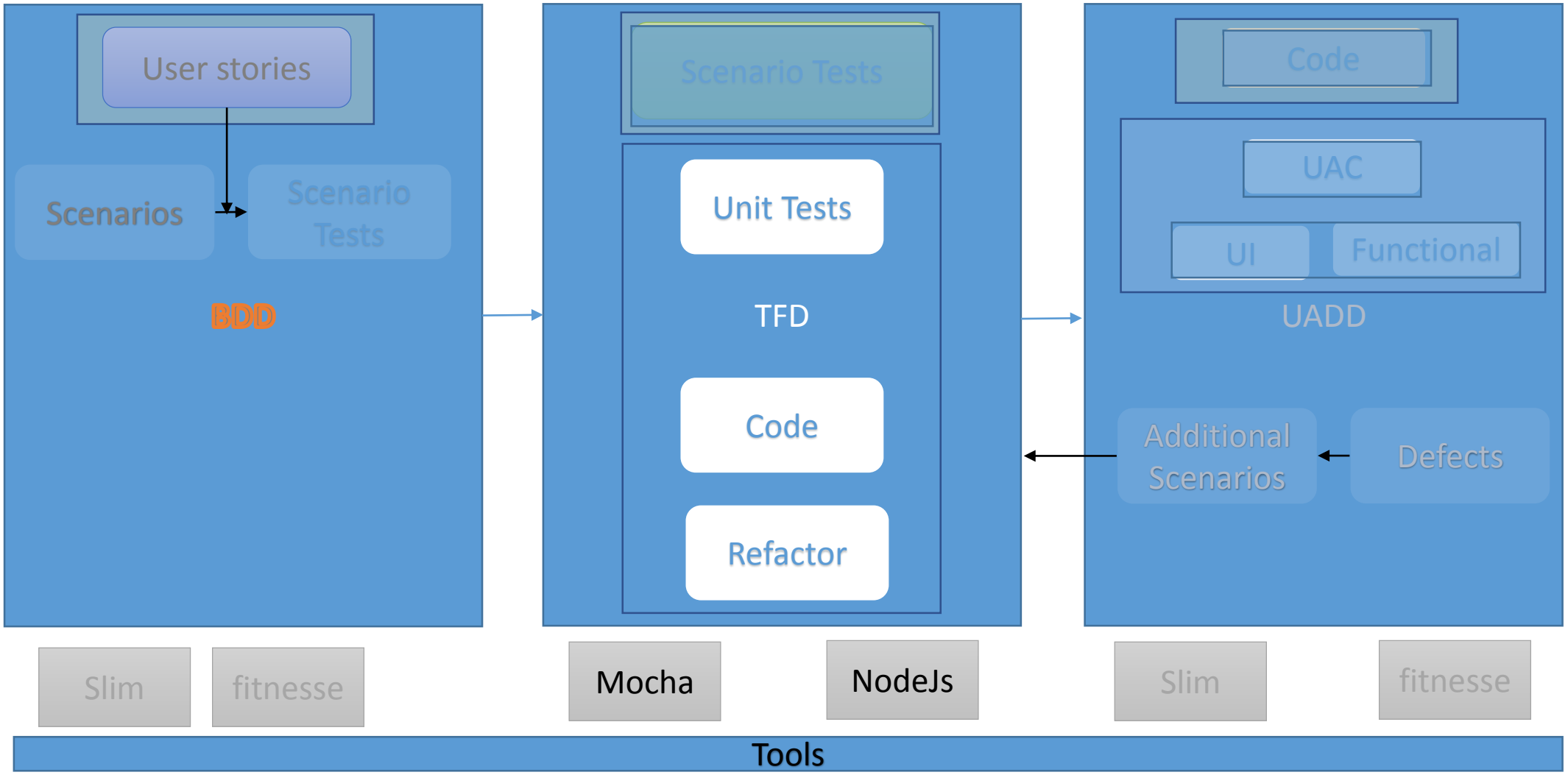
TDD with SlimJs

Presented

By

Ravichandran J V

BDD + TFD + UADD


User stories
Scenarios
Scenario Tests
BDD
Scenario Tests
Unit Tests
TFD
Code
Refactor
Code
UAC
UI
Functional
UADD
Additional Scenarios
Defects
Slim
fitnesse
Mocha
NodeJs
Slim
fitnesse
Tools

Given, when, then … in action!

Requirement

- User types some English sentence that must comply to a supplied drill line of text.

- If user misses or adds any additional words from the drill text, then the words must be highlighted as errors for the user to correct.

Given, when, then … in action!

Requirement

Given

- User types some English sentence that must comply to a supplied drill line of text.

when

then

- If user misses or adds any additional words from the drill text, then the words must be highlighted as errors for the user to correct.

Identify Scenarios (Behavior)

Scenario 1 – if user misses words

- If user misses or adds any additional words from the drill text,

Scenario 2 – if user adds words

Scenario testing

```
!define TEST_SYSTEM {slim}

!define COMMAND_PATTERN {node E:\fitnesse\node_modules\slimjs\src\SlimJS %p}

!path E:\fitnesse\slimtests

|import|
|test1 |

!|scenario|Missed word|

!|scenario|Added word|

|script|test1.Test1            |missing word|word|
|Missed word
|check |missingwordscenariotest|missing

|script|test1.Test1          |added word|added word added|
|Added word
|check |addedwordscenariotest|added
```

Scenario 1

Scenario & method

import
test1

scenario	Missed word

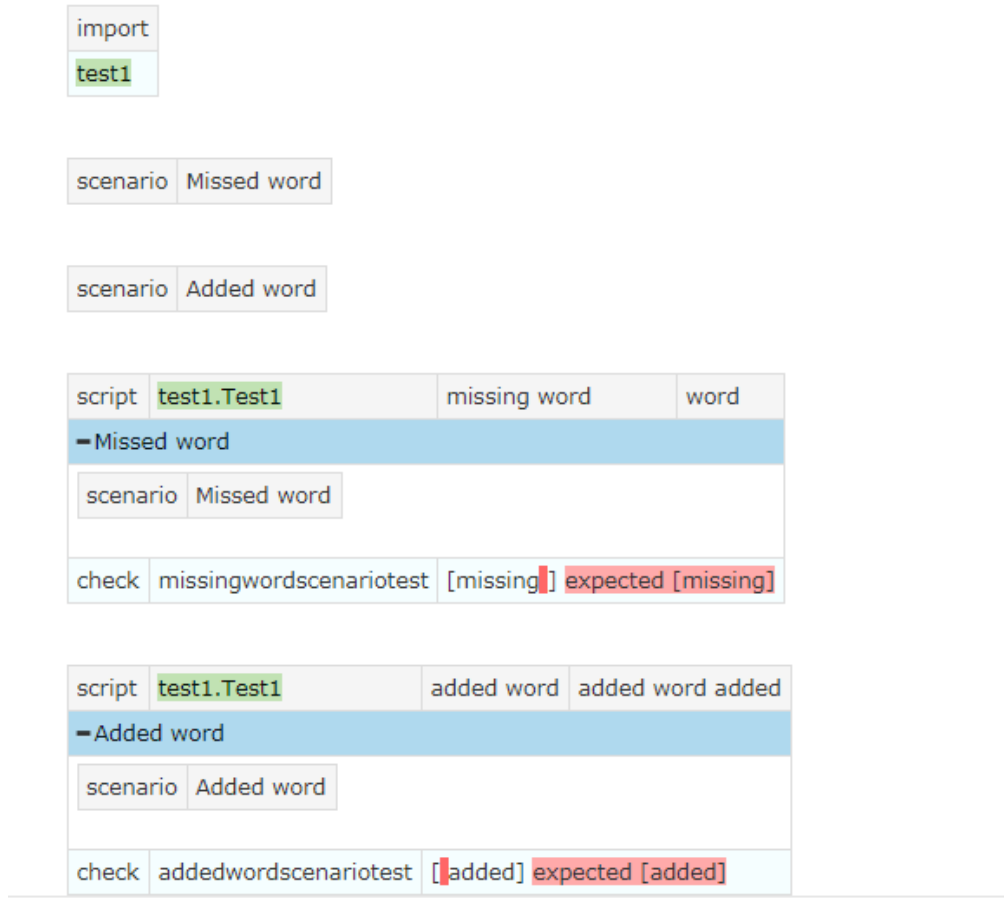
scenario	Added word

script	test1.Test1	missing word	word
-Missed word			
scenario	Missed word		
check	missingwordscenariotest	[missing]	expected [missing]

script	test1.Test1	added word	added word added
-Added word			
scenario	Added word		
check	addedwordscenariotest	[added]	expected [added]

Scenarios to Unit Tests

- The need for scenario tests becomes clear when we derive unit tests out of the scenario tests.
- Scenario tests enable an additional level of granularity to the unit tests apart from increasing traceability of business requirements/rules to implementation and back.
- In the scenarios analyzed in the previous step, BDD, the space before or after the missing word and the added word would not have been apparent if we had simply done a unit test!



Test Driven Development

- Test Driven Development (TDD) is done using unit testing
- A Unit test isolates and verifies individual units of source code
- A Unit Test is written first and if no code exists to satisfy the test, the unit test fails, always, in TDD
- A Unit Test drives development thus enabling design to evolve through satisfying the unit tests

Mocha – Unit test

Scenario – get missing word

```
describe('#get missing word', function() {
        it('should return missing word',function(){
        assert.equal(missingwordscenario(),"missing ");
        });
});
```

No code yet, so the test fails!

Add just the code to make the pass ie., add the method to run the assertion.

Mocha – Unit test

```
// twoscenarios.js
var jsDiff = require("diff");
function myApi(){};
myApi.prototype = {
twoScenarios : function(drill,user){
      var str="";
      var diff = jsDiff.diffWords(drill, user);
      this.missingwordscenario=function(){
          diff.forEach(function(part){
          if(part.removed){
             str+=part.value;
          }
          });
      return str;
   };
```

```
// test/Missingaddedwordtests.js
var app=require('.src/twoscenarios.js');
var assert=require('assert');

describe('#get missing word', function() {
     it('should return missing word',function(){
          var ob=new app.twoScenarios("missing word","word");
          assert.equal(ob.missingwordscenario(),"missing ");
    });
});
```

Mocha – Unit test

Scenario – get added word

```
describe('#get added word', function() {
        it('should return added word',function(){
                assert.equal(addedwordscenario(),"added");
        });
});
```

No code yet, so the test fails!

Add just the code to make the pass ie., add the method to run the assertion.

Mocha – Unit test

```
var diff1 = jsDiff.diffWords(drill, user);
  this.addedwordscenario=function(){
     str="";
        diff1.forEach(function(part){
        if(part.added){
           str+=part.value;
        }
        });
     return str;
  };

}
}
module.exports=new myApi();
```

```
// test/Missingaddedwordtests.js

describe('#get added word', function() {
  it('should return added word',function(){
     var ob=new app. twoScenarios("word","word added");
           assert.equal("added",ob.addedwordscenariotest());
});
});
```

Run the Tests

- Run the tests

```
tests>mocha missingwordtest.js


  #get missing word
    √ should return missing word

  #get added word
    √ should return added word


  2 passing (33ms)
```

- Check this

```
1) #get missing word should return missing word:

    AssertionError: 'missing ' == 'missing'
    + expected - actual

    -missing
    +missing
```

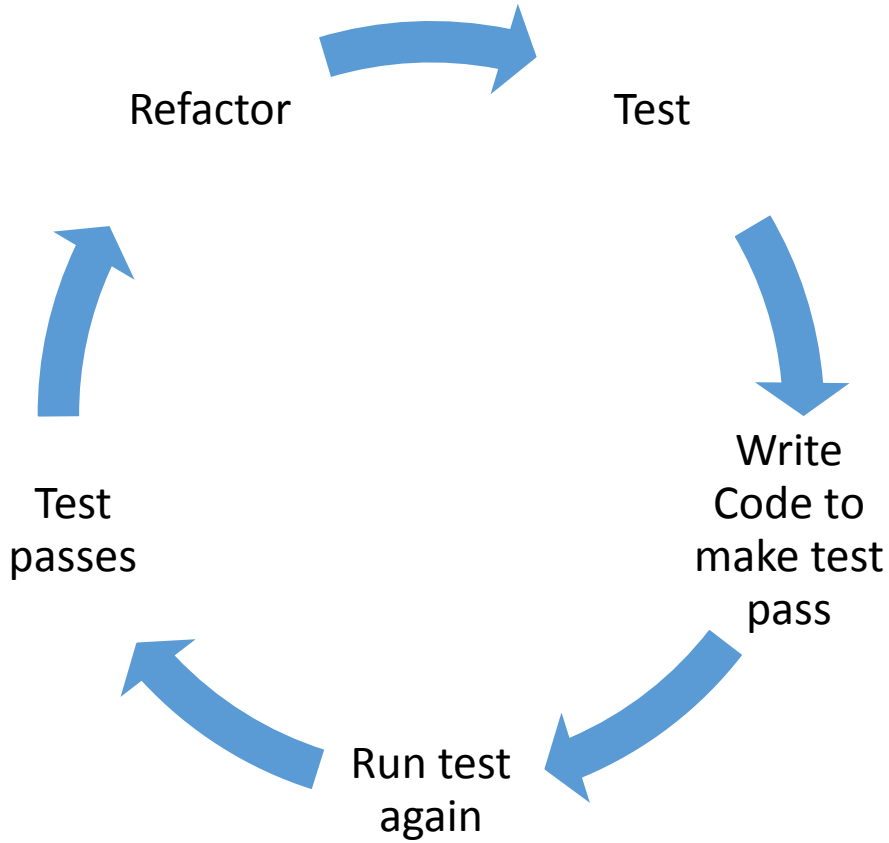
There is no space after “missing” and before “added” so the two tests fail:

```
2) #get added word should return added word:

    AssertionError: 'added' == ' added'
    + expected - actual

    -added
    + added
```

Best practices

- One assertion per test
- Refactor after test passes and run test again
- Add code only if the test result indicates the need
- Don’t add domain code in the test file. To ensure this practice, keep test files in ‘test’ folder and .js files in ‘src’ folder.
- Local variables like “str”, “diff”, “diff1” in this code need not be refactored!

TDD – Test, Code, Test Passes, Refactor Cycle



Thank you!