Advanced JavaScript

Syed Awase

0. PRE-REQUISITES AND SOFTWARE REQUIREMENTS

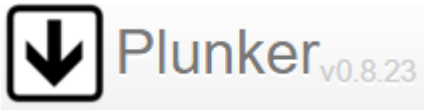
SOFTWARE REQUIREMENTS

Editors

Webstorm

Sublime Text

Visual Studio Code Editor



Additional Tools

Chrome Addons

Advanced Rest Client

ENABLE CORS

XHR Poster

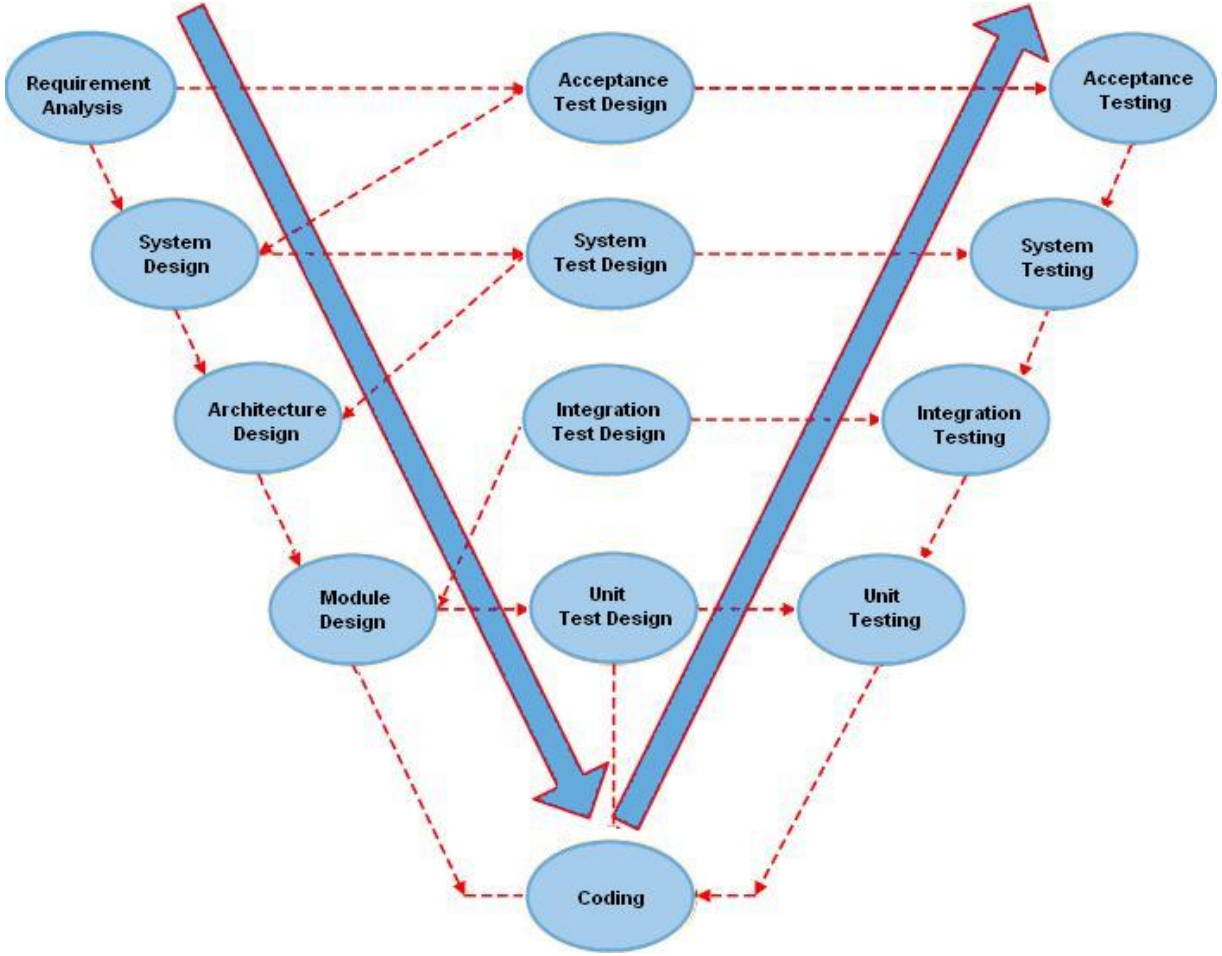
FORCECORS

1. UNIT TESTING WITH JASMINE

Types of Tests

Unit Test	•Single Class (usually) •No external dependencies
User Interface Tests	
Integration Tests	•Multiple classes working together •Will use a real database
Acceptance Testing	•Can Encompass “System” or “Functional” tests •End-to-End is desirable when feasible
Performance Testing	
Accessibility Testing	
Security Testing	
End-to-End Tests	•UI tests •Interact like a real user

Testing Life Cycle



Unit Tests

- Small pieces of code – public function or method
- Tests run in isolation (NO DEPENDENCIES)
- Fast to write and maintain
- Fast to run
- Run often during development
- Written by developer

Component Tests

- Test entire component in isolation
  - Any dependencies are still mocked out
- Developer's view of the world
- Tests more complex therefore harder to write
- Run slower
- Executed less often

Integration tests

- Interaction between units
- Minimize or use dependencies
- Internal units or internal and external units
- Harder to write and run
- Run slower
- Executed even less often

Performance Tests

- Time-based: How long does something take to execute?
- Size-based: How much data can the system handle?
- How long does the system take to respond?
- Can be at any level of system
- Difficulty to run, write, maintain depends
- Runtime depends
- Execution time depends
- Test the upper bounds of your application

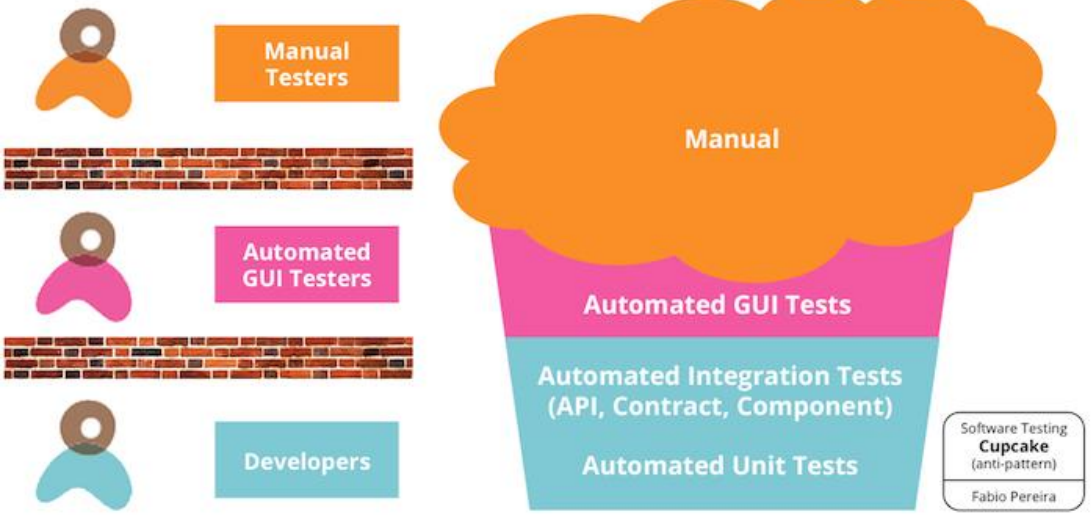
Feature Tests

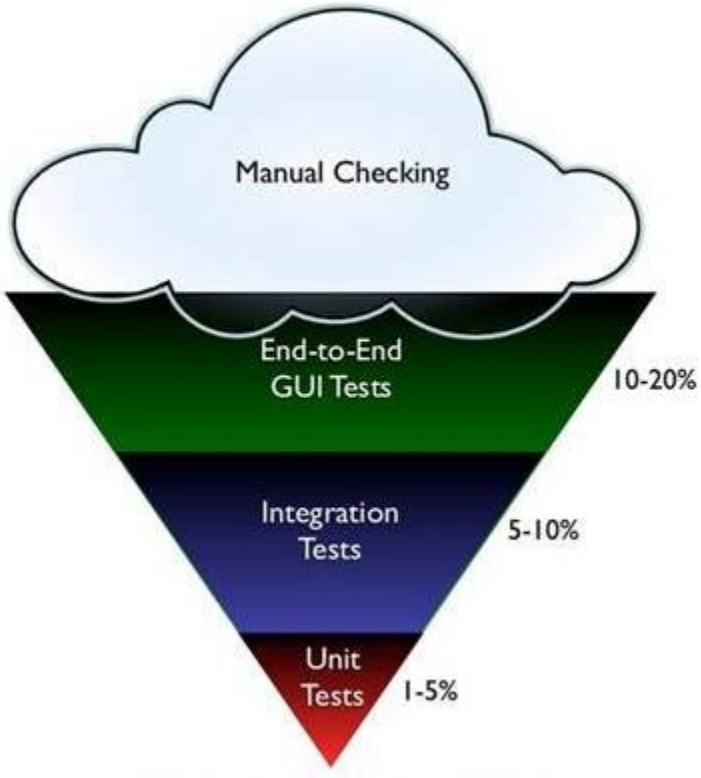
- User's view of the world
- Test feature
- Minimize or use dependencies
- Integration test-level difficulty to run and write
- Run Slower
- Executed with integration tests

ACCEPTANCE TESTS

- Suite of integration or feature tests
- Use real dependencies
- User's view of the real world – using 'user stories'
- Difficult to run, write and maintain
- Run slower
- Executed before Software Release

Software Testing ICECREAM CONE ANTI-PATTERN watirmelon.com





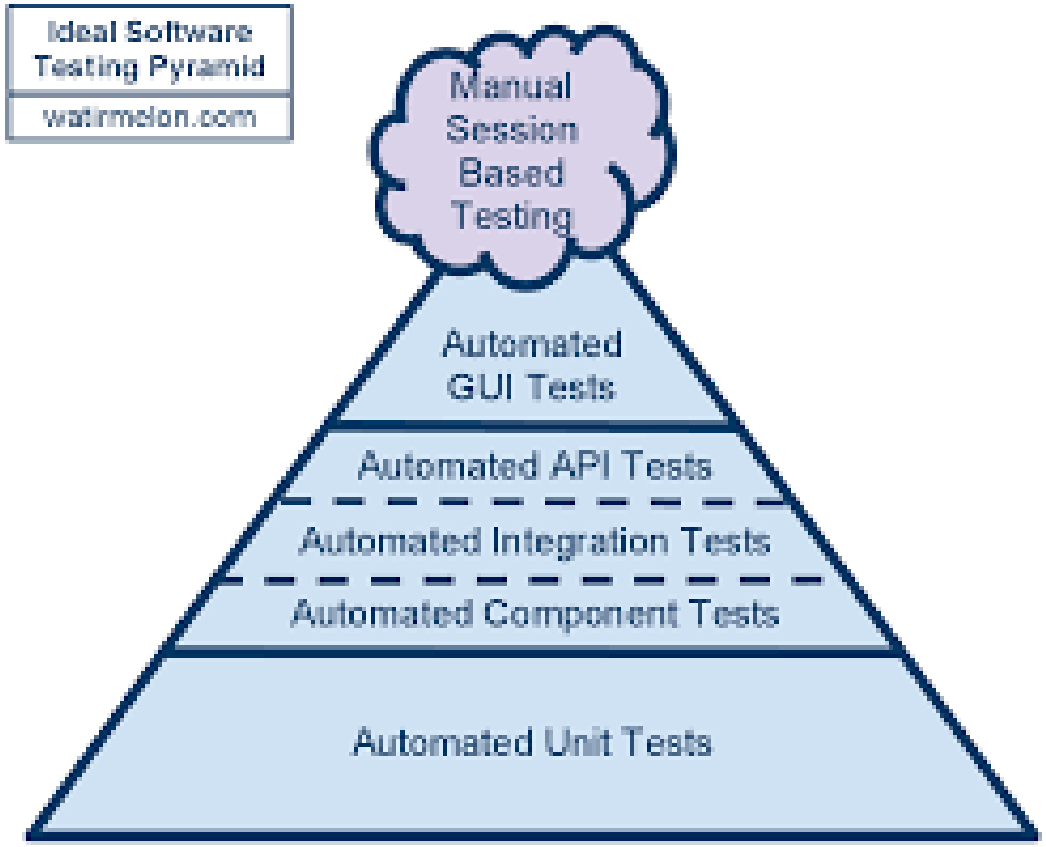
UI Tests

- Slow
- Brittle
- Hard to maintain; disconnect between developer and QA
- Non-deterministic; false negatives(test fails but it is not showing that there is a problem)
- Lots of moving pieces
- Hard to track down erros

TESTING PYRAMID

Mike Cohn(TEST PYRAMID)

" have many more low-level unit tests than high level end-to-end tests running through a GUI"



Unit Tests

- Fast
- Easy to maintain; written by developers
- Deterministic; no false negatives
- Minimal amount of moving pieces
  - Mocked out or stepped out any dependencies
- Error Isolation

- SO WHAT % of TESTING
  - **50% Unit Testing**
  - 10-15% Component Testing
  - 15-20% Feature Testing
  - 10% Integration Testing
  - 5% e2e
  - 5% manual

Unit Testing Environments

- Server Side

- Client Side
- Browser
  - Chrome, IE, Firefox
  - Headless browsers
    - Xvfb
    - PhantomJS and CasperJS
    - SlimerJS

ASPECTS OF UNIT TESTING

- Synchronous unit test vs. Asynchronous unit tests
- Fixtures
- Spies
- Mocks/Stubs
- Assertions
- AJAX
- Time
- Code Coverage

THIRD PARTY FRAMEWORKS

Other Tools

- Angular
  - ngMock
  - Protractor
- JSHint and ESLint
- Sinon Assertion Framework
- WebDriver
- Is.Js

AUTOMATION TESTING TOOLS

- KARMA
- JENKINS
- GRUNT
  GULP
- BOWER
- YEOMAN
- NPM

ASSERTIONS

- Functions that verify values are what you think they are
- Basis for almost all testing
- Write your own or use a library
- Standard ones come with Node.js/Framework
  - Existence /True/False
  - Type Checking
  - Throws

```
function assert(value,description){
    if(!value){
    throw new Error(description + 'isFalse!');
    }
}
```

```
console.log(5+6)
console.log(assert(5+6 === 12, '5+6 is11!'))
```

ASSERTIONS

- Assertions form the backbone of unit tests
- They are functions
  - Do nothing if a value is expected
  - Complain if value is unexpected
  - Easy to write and reuse
  - Basic ones come with Frameworks
  - NodeJS provides “assert” module
  - Used in almost all test types

- Basic Assertions
  - This is the only assertion function you will ever need!

```
function assert(value,description){
    if(!value){
    throw new Error(description + 'isFalse!');
    }
}
```

- Expressive Assertions
  - Convenience assertions
  - ==assert

NODEJS ASSERTIONS

var assert = require(“assert”);

assert.equal, assert.notEqual (==)

assert.strictEqual, assert.notStrictEqual(===)

assert.deepEqual, assert.notDeepEqual

assert.throws, assert.doesNotThrow

```
function assertThrows(func, desc){
    try{
    func();
    throw new Error(desc);
    }catch(e){

    }
}
```

```
assertThrows(function(){
        throw new Error('error');},"This throws an error!"

});
```

Type Assertions

- Type based assertions
  - ‘number’(integer, floats, NaN)
  - ‘boolean’ (‘true’ and ‘false’)
  - ‘string’
  - ‘object’ (Objects, Arrays and null)
  - ‘function’
  - ‘undefined’ (never defined or var ‘x’)
  - ‘symbol’(ES6)

- Object types
- Can my object be used whenever another object is expected?
- Is my object a sub-class of another object?
- Instanceof? Only useful with constructor functions and not clobbering prototype.constructor
- Duck typing? Does the object fullfill the object contract? Possible semantic mismatch?
- Inheritance vs. compostion – pick one style and stick to it.

Time based Assertions – Synchronous/Asynchronous

- Did Something complete in time?.
- assertInTimeSync()
- Asynchronous functions
- assertInTimeAsync()

SIDE EFFECTS

- Assertions typically test return values
- May also need to test DOM or CSS state
- May also need to test function metadata (spies)
  - Number of times called
  - Arguments

ISTANBUL CODE COVERAGE

CODE COVERAGE

https://gotwarlost.github.io/istanbul/

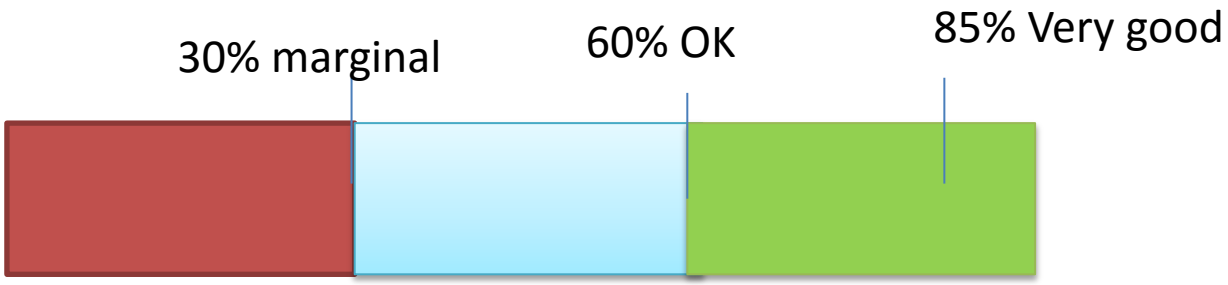
- Code coverage is an important metric in measuring unit testing reach.
- Simply measures how many times <code is executed>
  - Line (how many times a line has been executed?)
  - Statement (how many times has statement been executed?)
  - Branch – how often a branch has been executed?)
  - Function – How many functions have been executed?

- Coverage tools insert statements to record activity
- Only done for code you are testing
- Slows down execution
- Client side results must be beaconed back to the server
- Raw results converted to pretty HTML

Measure unit test coverage

- Typically measure code coverage of unit tests
- Higher coverage the better! Right?
- Trivial to write useless unit test to increase coverage

- Meaningful vs. Useless
- One measure of unit test effectiveness
- More complex code requires more unit tests.



Typical test flow

Phases

1. Global setup
2. Before suite
3. Before test
4. Test
5. After test
6. After suite
7. Global tear down

1. Global Setup

A. HTML Loading all initial dependencies
   A. Javascript under tests
   B. Javascript tests
   C. 3rd party tools/frameworks
B. Kickoff tests

Typical test flow

Before Suite

1. Setup suite-wide mocks/stubs/ fixture/handlers
2. Setup suite-wide reporting
3. Install mock ajax with spies
4. Create suite-wide HTML
5. Create suite-wide event handlers
6. Setup individual test reporting

Before Test

- Setup more specific mocks/stubs/fixture/handlers
- Install spies for every test
- Create test-focussed HTML
- Create test-focussed event handler

Typical test flow

During Test

- Load fixture
- Setup mocks/spies/stubs/listeners
- Execute method
- Assert return/timeout?
- Assert side effects
- Teardown mocks/spies/stubs/listeners
- Teardown fixture

After test

- Teardown more specific mocks/stubs/fixtures/handlers
- Remove spies for every test
- Remove test-focused html
- Remove test-focussed event handlers
- Complete individual test reporting

After Suite

- Teardown suite-wide mocks/stubs/fixtures/handlers
- Complete suite-wide reporting
- Remove mock ajax with spies
- Remove suite-wide HTML
- Remove suite-wide event handlers

Global teardown

- Collect and persist test results
- Collect and persist code coverage data
- Collect and persist any other metrics
  - Timings
  - Network data
  - Any other metadata

Manual Unit Testing

- Discover / Run all tests
- Collect all the results
- Process and display results
- Assertions
- Mocks and spies
- Synchronous and Asynchronous test support

- From Function name fooTest(){}
  - No ordering
  - Hacky
  - Unstructured – pollute global namespace
- Test call a global “test” method
  - Test(function(){}
- Generate HTML for client-side
- Generate runner for server-sde

- Collecting Results
  - What is a failure?
    - Return value – no explanation for failure
    - Thrown exception – better
- Capture Function name and result?
  - Suites have hierarchies of suite name +test name
- Stop on failure or continue?

- Process and Display results
  - Output results where
    - Console log –easiest
    - Persist back to server – need to add server hooks
    - Dump to file – easy once on server
    - What does the user see?

ASSERTIONS in MANUAL TESTING

- How many of them are you going to provide for testing?
- How many of them are detailed assertions?
- How are they included and used?

- MOCKS AND SPIES
  - Provide and support for these?
- Mocking
  - Each object/class provides mocks/stubs
  - Up to test writers to provide their own mocks/stubs
  - 3rd party mocking frameworks available
- 3rd party spy framework vs. Writing wrapper functions yourself.

SYNC AND ASYNC SUPPORT IN MANUAL TESTING

- Synchronous tests
  - Timeout support?
  - Call and wait?
- Asynchronous
  - Need callback functions for everything
  - Timeout support

3RD PARTY TESTING FRAMEWORKS

WHAT DO 3rd PARTY TESTING FRAMEWORKS PROVIDE?

- Hierarchical test organization and ordering
- Synchronous and Asynchronous test support
- Reporting
- Assertions
- Mock and Spy Support
- Client and Server-side test runners
- Support
- Updates

3rd Party TEST FRAMEWORK

- Learning curve
- Lose some flexibility
- Tied to tool
- May go away
- updates

- Jasmine
- Mocha and chai
- YUI
- Buster.js
- Protractor
- QUnit

- Nested suites of tests using **"describe"**
- **Asynchronous support**
- Set of included assertions **("matchers")**
- Nested setup and teardown
- Robust Spies
- Clock mocking (time based testing)
- Built in support for both server side and client side unit testing
- Ability to write customized matchers/assertions.
- Pretty reporting in a lot of different formats.

MOCHA

- Nested suites of tests using ‘describe’
- Async and Promise support
- Code coverage
- Pretty reporting and notifications based on the OS you are running.
- Setup and teardown
- Bring your own assertion library
- File watching

QUNIT

- Born out of Jquery and was used to test Jquery code
- Hierarchical test structure using “Qunit.module”
- Built-in assertions
- Asynchronous tests
- Event triggering
- HTML fixture support
- Execute previously failed tests before other tests

JSUNIT

- Similar to or a clone of Junit Clone
- Hierarchical test structure via suites
- Junit XML output
- Supplies test runner and server
- Setup/teardown
- Assertions
- Test functions must being with “test”

RUNNING TESTS

Client Side Execution

- Load HTML into browser
- Tests run in separate <iframe> or window
- Pre-bundled code
- Pre-minified code
- Refresh to re-run
- Need HTTP server
  - Browser and code on different machine
  - Persist/display results (include code coverage)
  - AJAX
  - Live reload
- Use simplest HTTP server possible
  - Some frameworks provide one(karma, JSUnit)

Server Side Execution

- Single script to find and execute all tests
- Tests run in separate NodeJS VM
- Re-execute script on file change
- Selective test execution based on file change

TEST ENVIRONMENT

- Standard Desktop browsers
  - Manually starting and loading tests
- **Auto start via launchers (Karma)**
- Selenium/Web Driver
- Run headless via xvfb on linux and mac
- Sauce labs /BrowserStack

https://saucelabs.com/

STANDARD MOBILE BROWSERS

- Chrome on iOS and Android
- Android Browser
- Safari
- Internet Explorer
- Firefox on iOS and Android
- Dolphin HD
- Opera

SCRIPTABLE BROWSER

- PhantomJS uses WebKit
- HtmlUnit uses Rhino + partial Rendering
- TrifleJS uses IE+PhantomJS API
- Zombie.js is a NodeJS browser +Assertions
- SlimerJS uses Gecko as a rendering engine
- NOT headless
- Working towards API compatibility with PhantomJS

Starting Tests

- Manually start browser and load HTML
  - One-shot and capture
- Use Web Driver to launch browser and load HTML
  - Selendroid and ios-driver
- Use built-in launcher to launch browser and load HTML
  - Karma and JSUnit

NodeJS

- Command line test execution
- Isolated from different OS versions
- Specify NodeJS version in package.json
- Don’t hardcode path separator
  - path.join
  - path.sep

EXECUTE NOW OR LATER?

SYNCHORNOUS EXECUTION

- Executes now
- Return error value
  - Return -1 or {error:'fail'}
- Throw Error Object
  - No problem just throw it
- When is it executed
  - It is executed before the next statement

ASYNCHRONOUS EXECUTION

- Executes later
- Right after the current code finishes execution
  - setTimeOut(function() {},0);
  - Process.nextTick(function() {});
  - Some amount of time from now
    - **setTimeout(function(){}, 3000);**
    - **setInterval(function(){},8000);**
    - When something happens
      - Element.addEventListener('click', function() {});
- Return Error value
  - Call back return error string/object
- Throw Error Object
  - **Wrap in a Promise**

Unit Testing

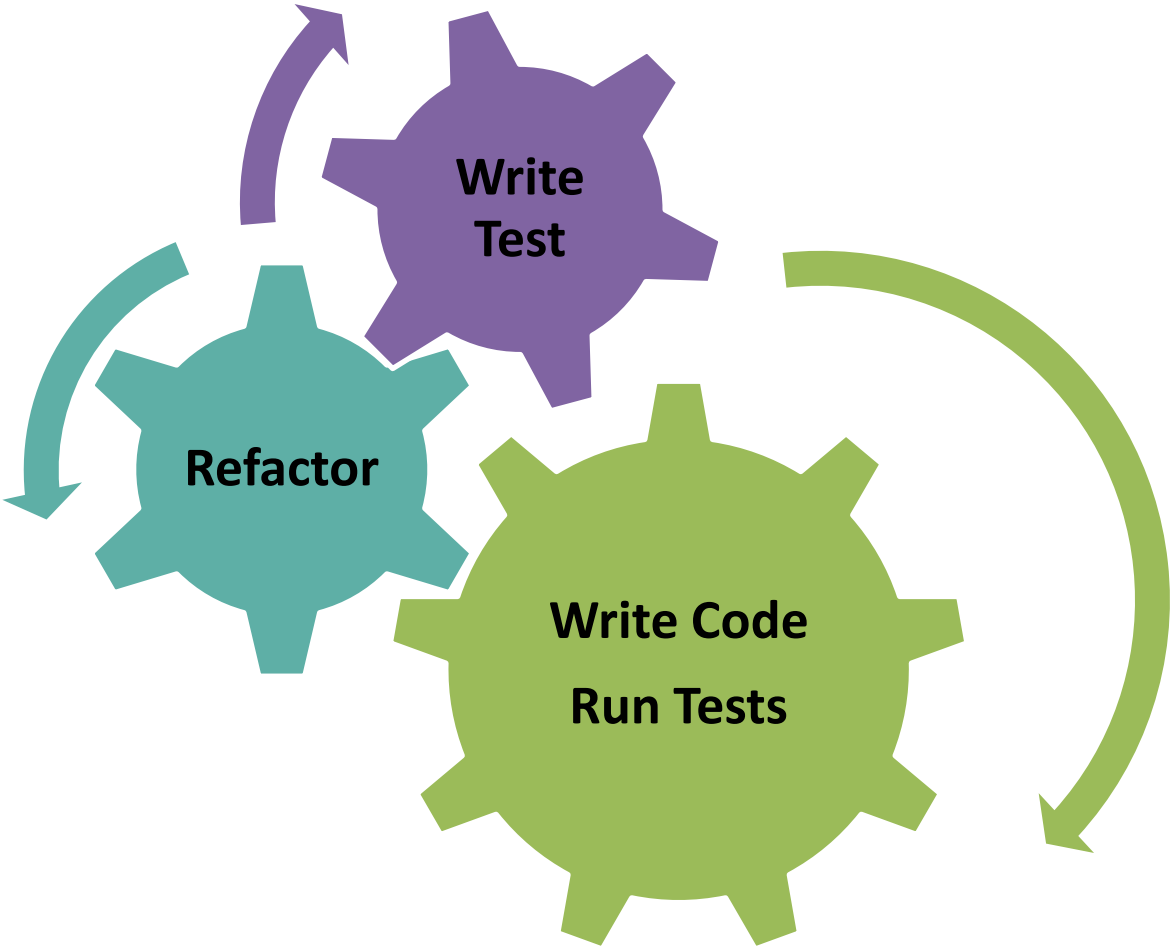
Synchronous

- Synchronous return value
  - Assert return value in the next statement
- Synchronous throw Error
  - **Wrap in try/catch**

ASynchronous

- Asynchrnous callback
  - Check error and assert value in callback
- Asynchronous throw Error
  - **Use promises**

Test Driven Development



- More Tests
- Great Coverage
- **"Testable" Code**
- Get the API right
- Easy to write tests
- Focus on the small scale
- Forces slowing down
- More boilerplate initially
- Longer to get started
- need to know where you are going

EACH TESTABLE UNIT

- Small (less code-less bugs, fewer tests, easy to test)
- Minimal dependencies (less complexity, easier to isolate, easier to test)
- Dependencies constructor-injected (swap implementations (replace real dependencies with mock dependencies), Easy to mock and test
- Program to interface – write tests once - many different implementations, defined contract
- Low complexity
- Minimal hierarchy (Inheritance vs composition)

MEASURING COMPLEXITY

- Number of Operands and Operators
- Lines of code
- Amount of comments
- Cyclomatic complexity
- Halstead values
- Maintainability index
- Fan in
- Fan out

CYCLOMATIC COMPLEXITY

- An integer – is the number of independent paths through your code
- Number of unit tests for 100% coverage
- **Originally proposed by Thomas McCabe(1976)**

```
function sum(a,b){
        switch(typeof a){
    case "number": return a + parseInt(b,8);    }
    case "string":{return a+b;}
    case "object":{throw new Error('object?');}
    }
}
```

Cyclomatic complexity is 4

Best Practices

1. Keep the value below 10.
2. Lots of factors
    a. Understand ability of the code
    b. Testability
    c. Complexity
    d. Simplicity
3. Refactor into other methods

HALSTEAD METRICS

- Volume of Code = Amount of Operands and Operators
  - X= Z+1
- Difficulty Score = How difficult is the program to write or understand.
  - Unique operators and operands and total operands.
- Effort = Difficulty Score x Volume of Code
- Time required = (Effort/18)seconds
- No of Bugs = (Halstead Volume /3000)

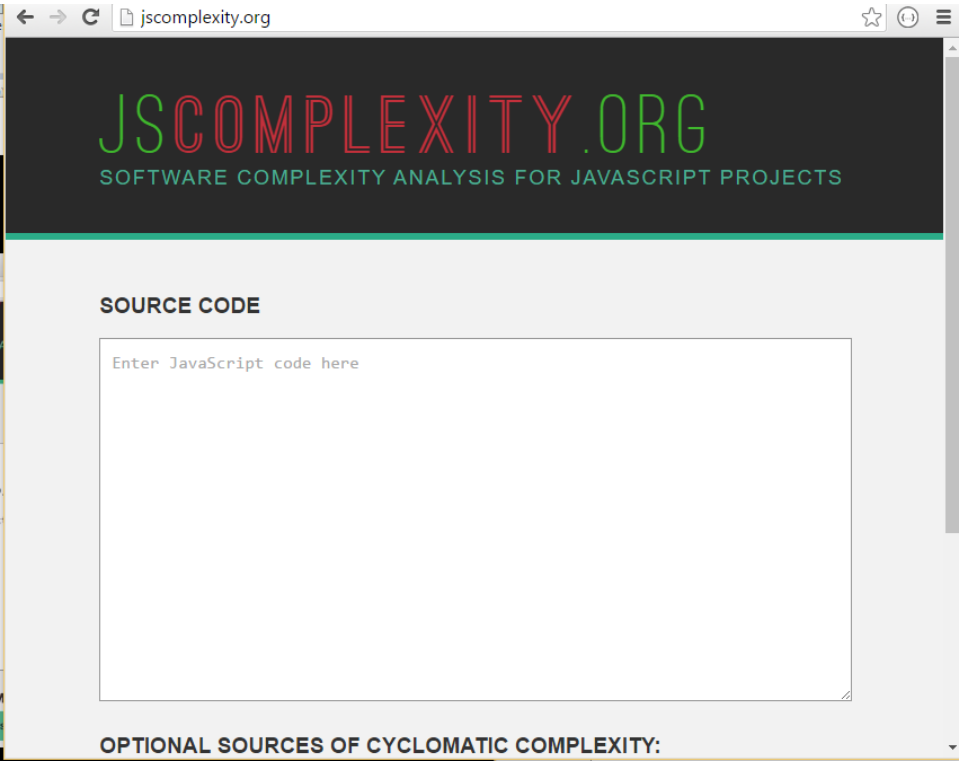
MAINTAINABILITY INDEX

- How easy is it to maintain a block code?
- Built on top of CYCLOMATIC COMPLEXITY and Halsted Metrics

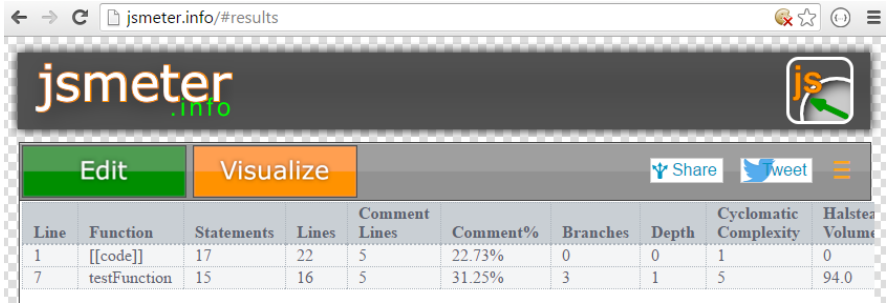
MI = 171 -5.2*ln(Halstead Volume) -0.23*(Cyclomatic complexity) -16.2* ln(Lines of Code)

Microsoft Visual Studio has normalized the maintainability index between 0 to 100

SOFTWARE COMPLEXITY ANALYSIS

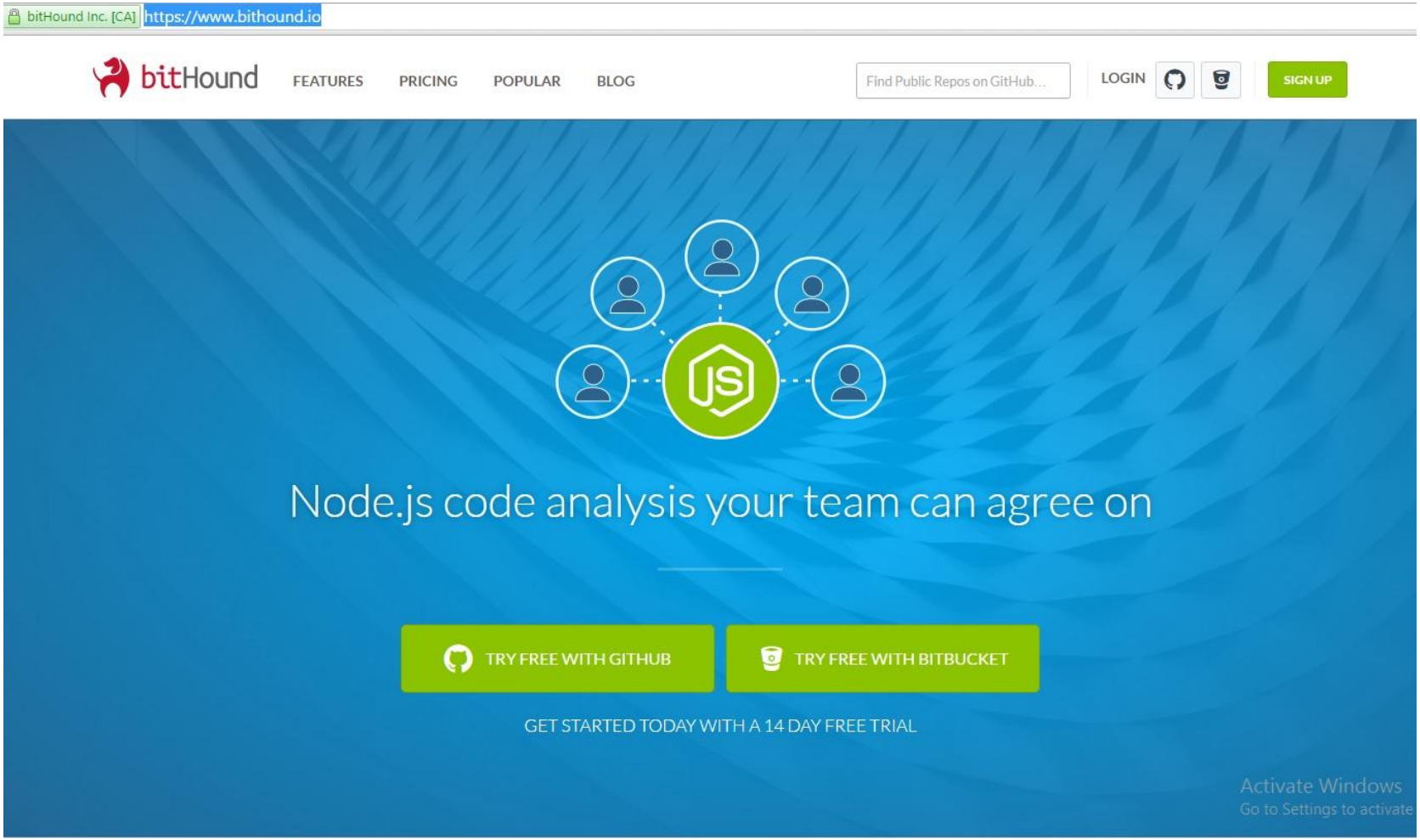


http://jscomplexity.org/



http://jsmeter.info/

https://www.bithound.io/



FAN OUT

- Number of dependencies of this unit
- Big Numbers bad
- Small Numbers good
- Speculation that more than 5 or 5 dependencies is bad
- Source of fan out
  - Parameter Lists
    - Constructor
    - Methods
  - Internal
    - Instantiated modules
    - Private
    - Properties

FAN IN

- Number of dependees of this Unit
- Big Numbers good
- Small Numbers bad
- Core modules should have maximum fan in
- Sources fan in
  - Injected into methods
    - Constructors
    - Methods
  - Tightly coupled dependencies
    - Created internally
    - Public or private properties

Coupling

Coupling defines how modules are related to each other

- Tight Coupling
  - Dependencies hidden in object
  - Possibly inaccessible
  - Dependencies instantiated locally
    - No control over scope
    - Object must manage dependent lifecycle
  - Must use that exact object
  - Not modular, not reusable
  - Hard to test
  - Bad design
- Loose Coupling
  - Dependencies required to build object
  - Dependencies are accessible
  - No dependencies instantiated locally
    - Full control over scope
    - Lifecycle managed by something else
  - Use whatever object you want
  - Modular and reusable
  - Good design

Inheritance **vs** Composition

Inheritance

- Object trees
- Super classes, subclasses
- Use “new” function constructors and prototype property
- Information hiding
  - Public – “this” and prototype
  - Protected – use closure
  - Private – no references outside of scope

Composition

- Program to interfaces
- Mix and match functionality
- No function constructors or prototypes
- No “new” no function constructors, can use prototype
- **Usually easier to test**

WHAT IS A DECORATOR?

- Adds functionality to an already existing object
- Proper way to incorporate cross-cutting concerns in a single object
- Allows for single purpose interface (Single Responsibility Principle)
- Can keep decorating a single object

Relative values

Measuring complexity

- Effort to understand the code
  - Standard idioms
  - Cleverness
- Modularity
- Coupling
- Inheritance vs composition
- Number of tests and code coverage

- Code Reviews
- Test Coverage
- Ease of writing new tests
- Time to fix a bug
  - Current developer
  - New developer

TENANTS OF TESTABILITY

- Testability = maintainability
- Only one way to know is to try to write tests
- Writing tests is easy at the beginning
- Testable code makes writing tests later
- Non testable code is useless

- How long will it take to fix a bug?
- Who will find a bug?
- How long will it take to hand code over?
- How long will it take someone to fix your code?

SALIENT FEATURES OF TESTABLE CODE

- Small code blocks
- Modular/loosely coupled
- Do only one thing at a time
- Minimal side effects
- Consistent
- Proper Naming Conventions and Best Practices
- Obey Solid Principles
- Not complicated code blocks
- Compliance with Maintainability Index
- Create lots of small interfaces
- Constructor inject all dependencies
- Keep shallow object hierarchies
- Keep measuring

A behaviour driven JavaScript development tool

SYED AWASE KHIRNI

1A. JASMINE

BEHAVIOUR DRIVEN DEVELOPMENT

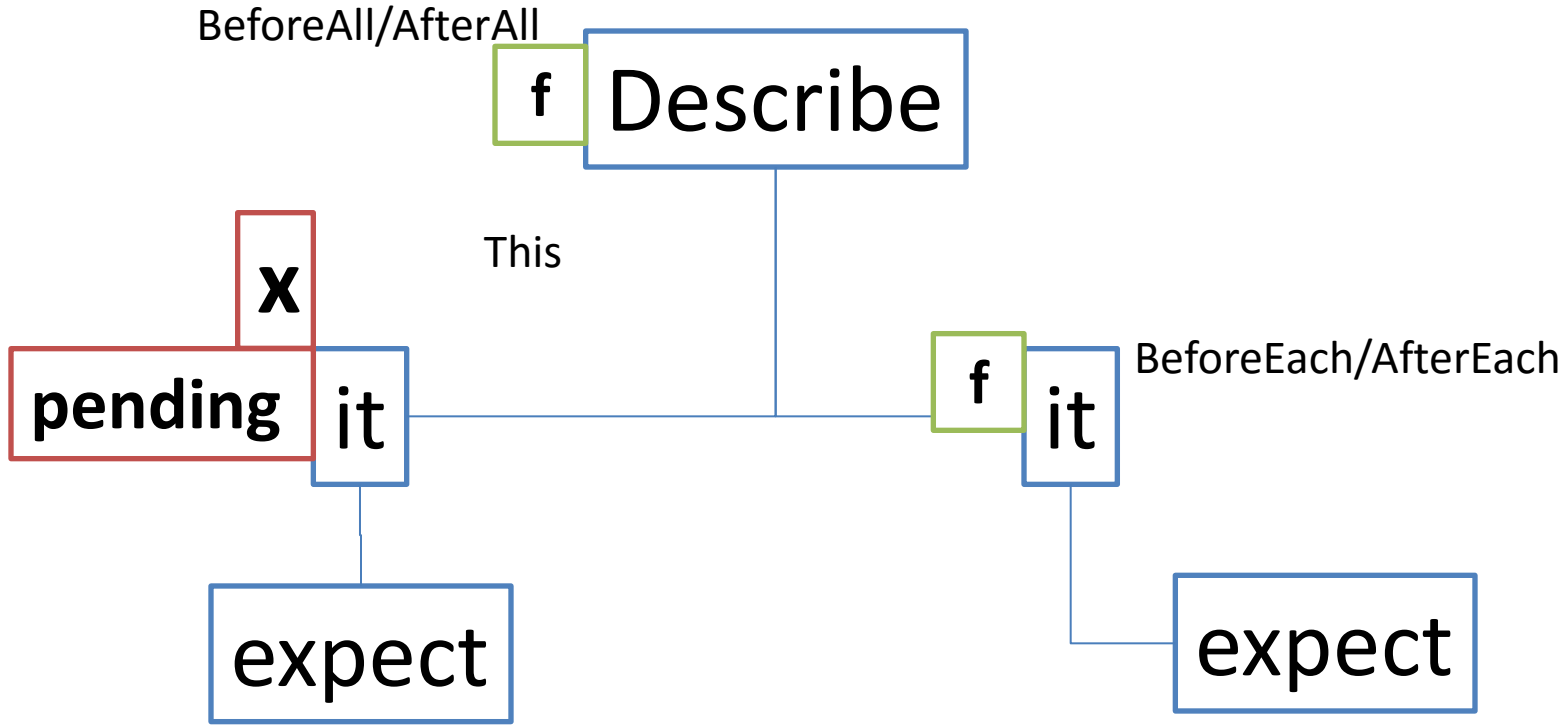
- More specific framing of Test driven development TDD
- Test desired behaviour
- Anyone can understand what is being tested
- Semi-formal test specification like user stories
  - Use “should” in test names
  - When/Then
- Natural Language Terminology used to describe testing
- Jasmine- Testing
- Cucumber –Development

- http://jasmine.github.io
- Current version 2.4
- It does not depend on any other JavaScript frameworks.
- It does not require a DOM
- Clean and Obvious Syntax to write tests.
- Suite
  - Begins with a call to global Jasmine Function **describe**
    - **With two parameters**
      - **A string and a function**

TEST GRAPH

Do not run these

Only run these



Matchers

- Expectations have matchers
- Expect(foo).toBe(bar);
- Some Matchers that come with jasmine
  - String
  - Existence
  - Object
  - Array
  - Custom

SPIES

- Spies are functions that track parameter list
- Track function invocation
- Parameter lists for all invocations
- Can call through
- Can call another function
- Can return arbitrary value
- Can throw Error

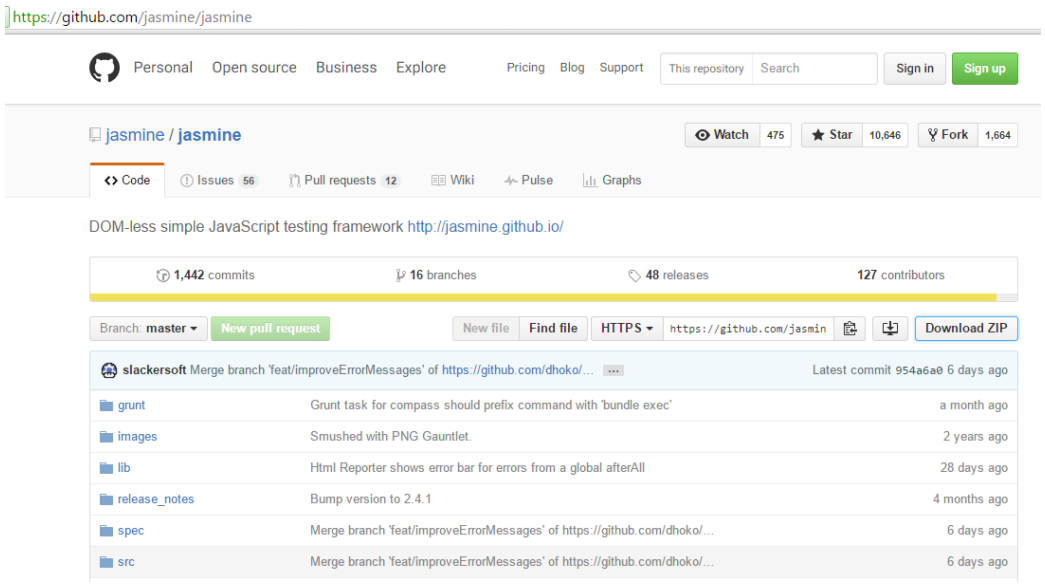
Mocking the Clock and Date

- Synchronous setTimeout/setInterval
- Set Date to anything

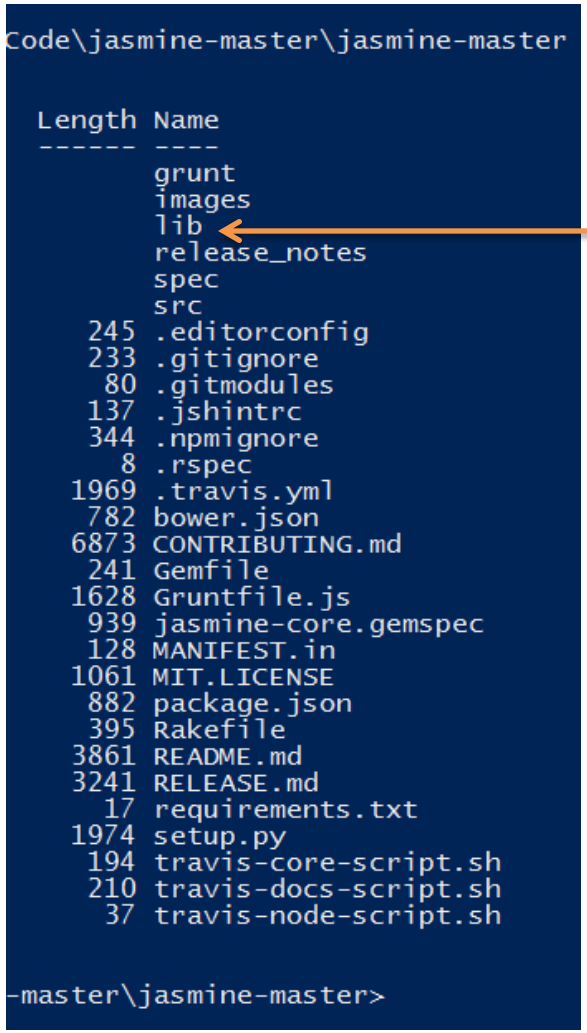
Asynchronous Support

- BeforeEach/afterEach
- beforeAll/afterAll
- It
- Default timeout 5 seconds
- Optional “done” parameter
- Done.fail() to fail test

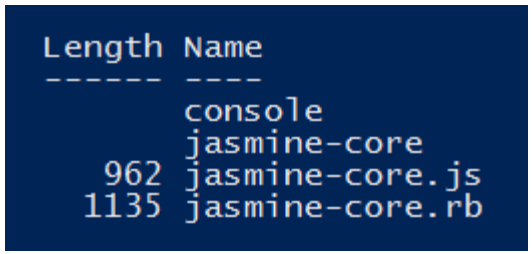
Jasmine standalone



lib



lib



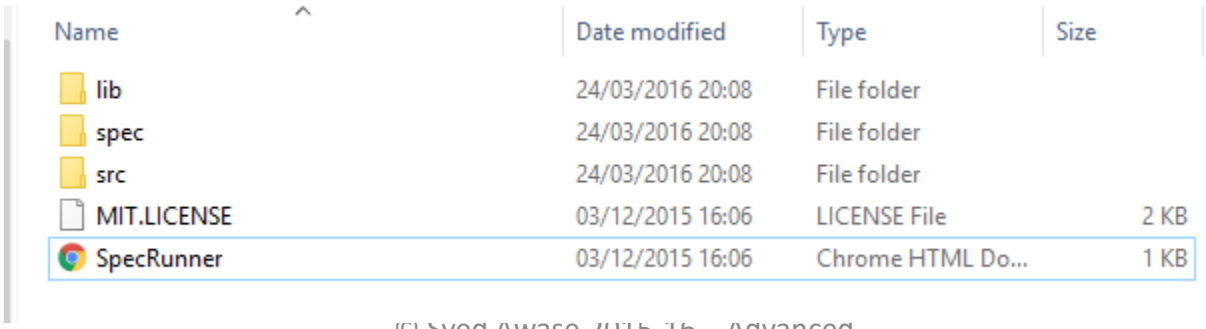
Setting up requirejs+jasmine+node.js

- Npm install –g jasmine-node
- Npm install –g requirejs

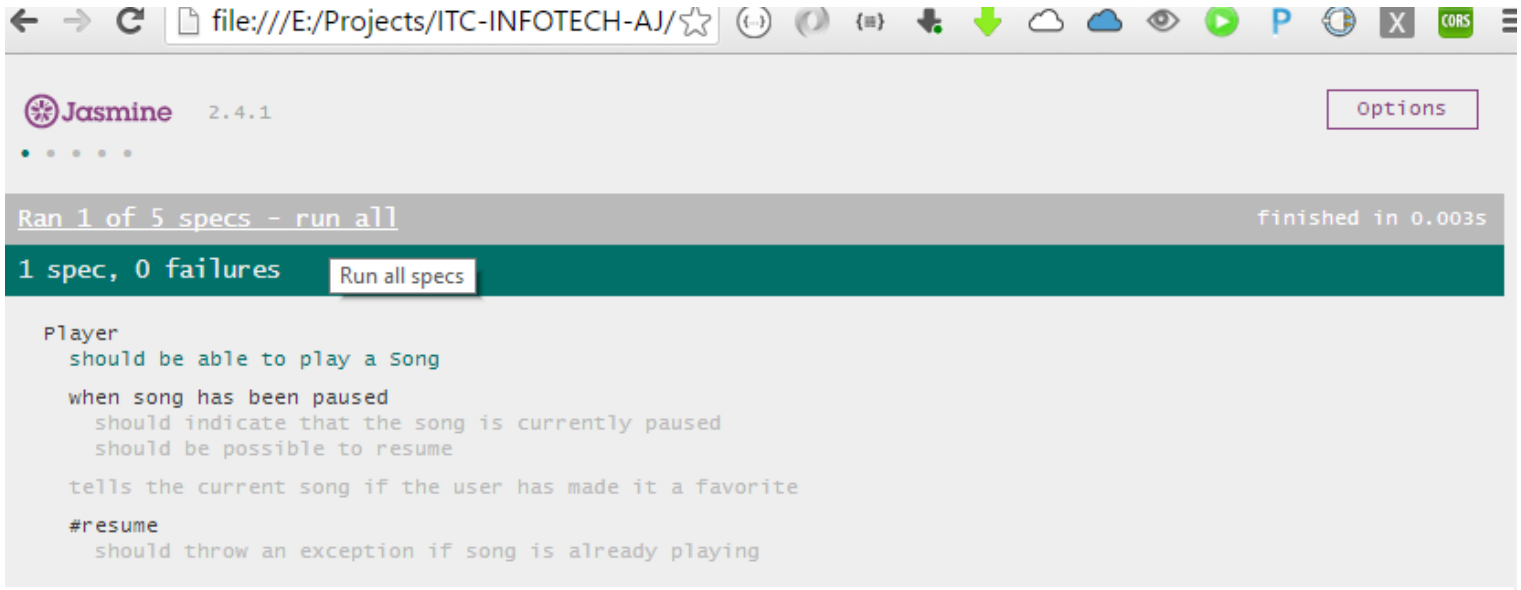
Standalone Version of Jasmine

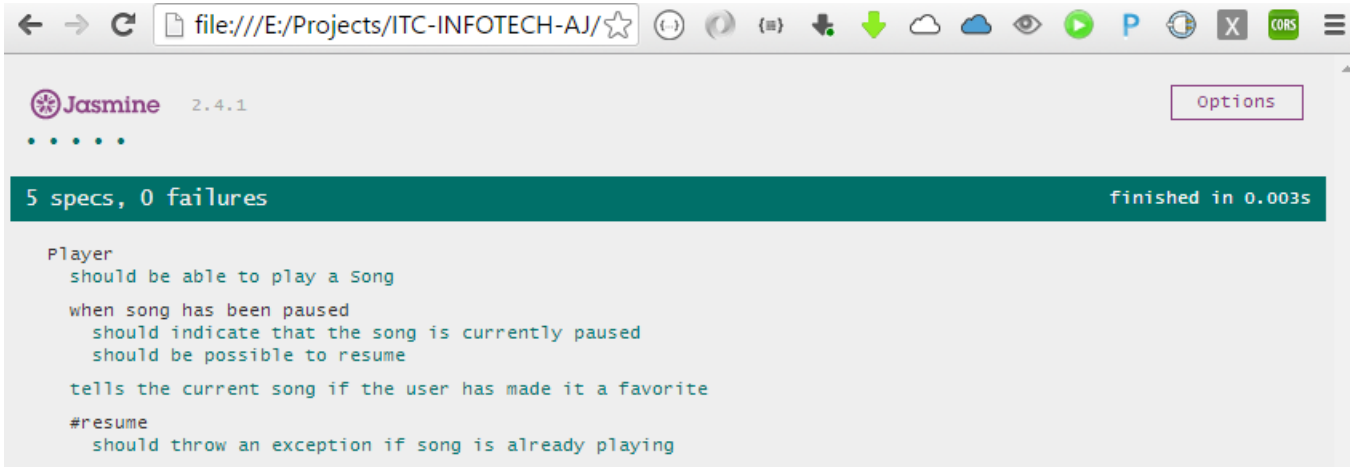
https://github.com/jasmine/jasmine/releases/download/v2.4.1/jasmine-standalone-2.4.1.zip

Extract it and open the file **SpecRunner.html in the browser**



SpecRunner





Server Side Installation of Jasmine

npm install –save-dev –g jasmine
(global installation of jasmine)

mkdir jasmine_test

cd jasmine_test

npm init

more package.json

npm install –save-dev jasmine

npm test

jasmine init

jasmine examples

npm test

```
Length Name
------ ----
       node_modules
   224 package.json
```

package.json

```
{
  "name": "jasmine_test",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "jasmine"
  },
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "jasmine": "^2.4.1"
  }
}
```

```
Length Name
------ ----
       node_modules
       spec
   224 package.json
```

HelloWorld Test

Spec/helloWorld_spec.js

```
describe("Hello World", function(){
    it("Should return hello world", function(){
    expect(helloWorld()).toEqual('Hello World');
    });
});
```

Src/helloWorld.js

```
var helloWorld = function(){
    return 'Hello World';
};
```

```
<!-- include spec files here... -->
 <script src="spec/helloWorld_spec.js"></script>


  <!-- include source files here... -->
<script src="src/helloWorld.js"></script>
```

Understanding the Spec

```
describe("Calculator", function(){
    it("Should store current value at all times",function(){
        expect(Calculator.current).toBeDefined();
    });
    it("Should add numbers", function(){
        expect(Calculator.add(5)).toEqual(5);
        expect(Calculator.add(5)).toEqual(10);
    });
});
```

.toBeDefined()
.toEqual()

Called as matchers or assertions

Calculator Example

Calculator.js

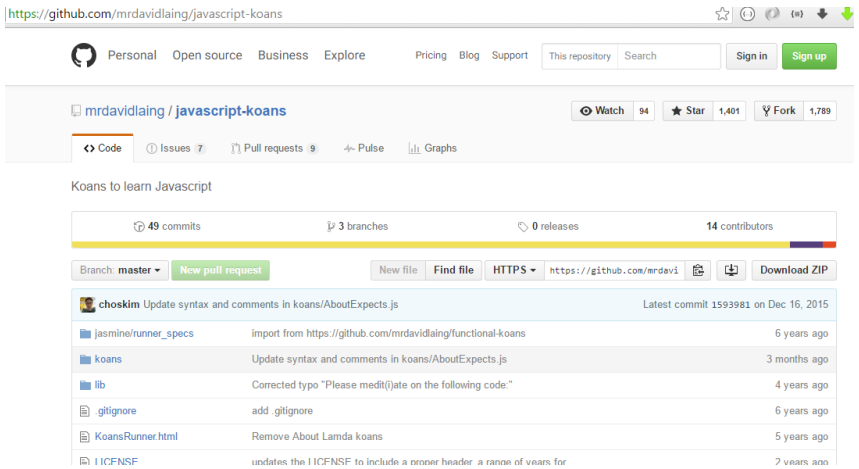
```
window.Calculator= {
    current :0,
add:function(number1){
    var sum = this.current;
for (var i = 0,len=arguments.length; i < len; i++) {
    sum += arguments[i];
}

this.current = sum;
// this.current+=number1;
 return this.current;
}
};
```

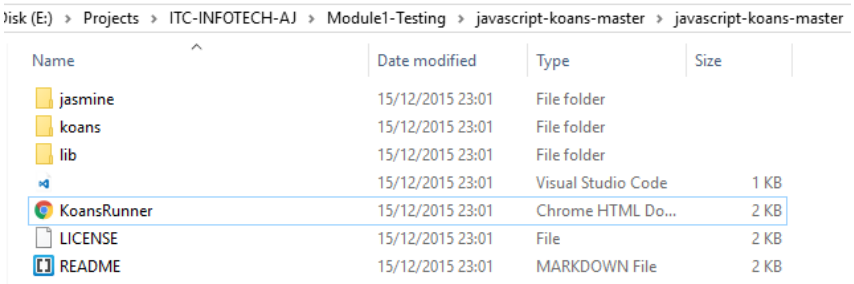
calculator_spec.js

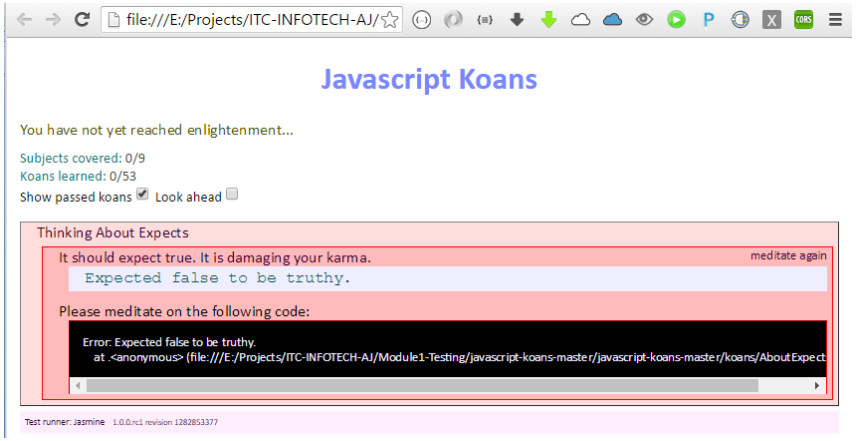
```
describe('Calculator', function() {
    beforeEach(function() {
        Calculator.current = 0;
    });
    describe("when adding numbers", function() {
        it("Should store current value at all times", function() {
            expect(Calculator.current).toBeDefined();
            expect(Calculator.current).toBeDefined(0);
        });
        it("Should add numbers", function() {
            expect(Calculator.add(5)).toEqual(5);
            expect(Calculator.add(5)).toEqual(10);
        });
        it("Should add any number of numbers", function() {
            expect(Calculator.add(1, 2, 3)).toEqual(6);
            expect(Calculator.add(1, 2, 3, 4)).toEqual(16);
        });
    });
     beforeEach(function() {
        Calculator.current = 0;
    });

});
```

JavaScript Koans

https://github.com/mrdavidlaing/javascript-koans







Please complete all the koans as exercise

JASMINE MATCHERS

toBe

- Uses strict equality (===) to compare the actual and expected values

API

```
expect(actual).toBe(expected);
```

Example

```
expect(12).toBe(12);    // pass
expect("12").toBe(12); // fail
```

toBeCloseTo

- Checks that numeric actual and expected values are equal up to a given level of decimal precision.
- If no precision is specified, it defaults to the value specified.

API

```
expect(actualNumber).toBeCloseTo(expectedNumber, precision);
```

Example

```
expect(2.00).toBeCloseTo(2.01, 1);  // pass
expect(2.00).toBeCloseTo(2.01, 2);  // fail

expect(3.00).toBeCloseTo(3.01);     // fail
expect(3.000).toBeCloseTo(3.001);   // pass
```

toBeDefined

- Checks that the actual value is defined (!==undefined)

API

```
expect(actual).toBeDefined();
```

Example

```
var a = 1;
expect(a).toBeDefined(); // pass

var b;
expect(b).toBeDefined(); // fail

var c = null;
expect(c).toBeDefined(); // pass
```

toBeFalsy

- Checks whether the actual value is falsy. A falsy value is one that can evaluate to false.

API

```
expect(actual).toBeFalsy();
```

Example

```
expect(false).toBeFalsy();      // pass
expect(null).toBeFalsy();       // pass
expect(undefined).toBeFalsy();  // pass
expect("").toBeFalsy();         // pass
expect(0).toBeFalsy();          // pass

expect(true).toBeFalsy();     // fail
expect("BMW Z3").toBeFalsy(); // fail
expect(1).toBeFalsy();        // fail
expect({}).toBeFalsy();       // fail
```

toBeGreaterThan

- Checks whether the actual value is greater than the comparison value

API

```
expect(actual).toBeGreaterThan(comparison);
```

Example

```
expect(2).toBeGreaterThan(1); // pass
expect(1).toBeGreaterThan(2); // fail
```

toBeLessThan

- Checks whether the actual value is less than the comparison value

API

```
expect(actual).toBeLessThan(comparison);
```

Example

```
expect(1).toBeLessThan(2); // pass
expect(2).toBeLessThan(1); // fail
```

toBeNull

- Checks whether the actual value is null

API

```
expect(actual).toBeNull();
```

Example

```
expect(null).toBeNull();        // pass
expect(undefined).toBeNull(); // fail
expect("").toBeNull();          //false
```

toBeTruthy

- Checks whether the actual value is truthy. A truthy value is one that can evaluate to true

API

```
expect(actual).toBeTruthy();
```

Example

```
expect(true).toBeTruthy();              // pass
expect("Aston Martin").toBeTruthy(); // pass
expect(1).toBeTruthy();                 // pass
expect({}).toBeTruthy();                // pass

expect(false).toBeTruthy();         // fail
expect(null).toBeTruthy();          // fail
expect(undefined).toBeTruthy();  // fail
expect("").toBeTruthy();            // fail
expect(0).toBeTruthy();             // fail
```

toBeUndefined

- Checks that the actual value is undefined

API

```
expect(actual).toBeUndefined();
```

Example

```
var a;
expect(a).toBeUndefined(); // pass

var b = 1;
expect(b).toBeUndefined(); // fail

var c = null;
expect(c).toBeUndefined(); // fail
```

toContain

- Checks an array to determine whether it contains the expected value

API

```
expect(actualArray).toContain(expectedItem);
```

```
Example
------------
var names = ["syed","awase","sadath"];
expect(names).toContain("syed"); //pass
expect(names).toContain("sadath");//pass
expect(names).toContain("ravi");//fail
```

toEqual

- Checks whether the actual and expected values are equal. For primitives like strings and numbers, strict equality(===) is used. For object and array, their members are compared with strict equality.

API

```
expect(actual).toEqual(expected);
```

Example

```
expect(1).toEqual(1); // pass
expect("1").toEqual(1); // fail
```

toHaveBeenCalled

- Checks to see if a spy function has been called.

API

```
expect(spy).toHaveBeenCalled();
```

```
Example
---------------
var raw ={
    agent: function(intel){
    return intel;
    }
}
spyOn(raw,"agent");
expect(raw.agent).toHaveBeenCalled();//fail
raw.agent();
expect(raw.agent).toHaveBeenCalled();//pass
```

toHaveBeenCalledWith

- Checks to see if a spy function has been called with a given set of **arguments.**

API

```
expect(spy).toHaveBeenCalledWith(arg1...);
```

```
Example
--------------
var raw ={
    agent: function(intel){
    return intel;
    }
}
spyOn(raw,"agent");
raw.agent("InvasionPlan");
expect(raw.agent).toHaveBeenCalledWith("WMD");//fail
expect(raw.agent).toHaveBeenCalled("InvasionPlan");//pass
```

toMatch

- Checks to see if the actual value matches the expected regular expression

API

```
expect(actual).toMatch(expectedRegex);
```

Example

```
var transmission = "coordinates: 125, 200";

expect(transmission).toMatch(/coordinates/); // pass
expect(transmission).toMatch(/plan/);        // fail
```

toThrow

- Checks to see if a function throws an exception

API

```
expect(theFunction).toThrow();
```

Example

```
function planA() {
  throw new Error("Kitten acquisition failed");
}

function planB() {
  return "Kitten acquired";
}

expect(planA).toThrow(); // pass
expect(planB).toThrow(); // fail
```

not

- Changing **not** between the expect function and the matcher function will **reverse the meaning of the matcher**

API

```
expect(actual).not.toBe(unexpected);
```

Example

```
expect("7").not.toBe(7);   // pass
expect("7").not.toBe("7"); // fail
```

toBeNull

- Checks to see if something is null

```
expect(null).toBeNull();                // success
expect(false).toBeNull();               // failure
expect(somethingUndefined).toBeNull();  // failure
```

toBeNaN

- Checks if something is NaN

```
expect(5).not.toBeNaN();                // success
expect(0 / 0).toBeNaN();                // success
expect(parseInt("hello")).toBeNaN();    // success
```

toBeCloseTo

- Checks if a number **is close to another number**, given a certain amount of decimal precision as the second argument.

```
expect(12.34).toBeCloseTo(12.3, 1);  // success

expect(12.34).toBeCloseTo(12.3, 2);  // failure

expect(12.34).toBeCloseTo(12.3, 3);  // failure
expect(12.34).toBeCloseTo(12.3, 4);  // failure
expect(12.34).toBeCloseTo(12.3, 5);  // failure

expect(12.3456789).toBeCloseTo(12, 0);   // success
expect(500).toBeCloseTo(500.087315, 0);  // success
expect(500.087315).toBeCloseTo(500, 0);  // success
```

toMatch

- Checks if something is matched, given a regular expression. It can be passed as a regular expression or a string, which is then parsed as a regular expression.

```
expect("foo bar").toMatch(/bar/);
expect("horse_ebooks.jpg").toMatch(/\w+.(jpg|gif|png|svg)/i);
expect("jasmine@example.com").toMatch("\w+@\w+\.\w+");
```

Custom Matchers

toBeLarge

- Checks if a number is greater than 100.

toBeWithinOf(n1,n2)

- Checks if a number is between two given numbers, inclusive
- It calculates the lower and upper bounds and the matcher's results is a simple bounds check.

```
// Expect 6 to be within 2 of 5 (between 3 and 7, inclusive).
expect(6).toBeWithinOf(2, 5);
```

Custom Match(**toBeWithinOf**)

```
beforeEach(function() {
    this.addMatchers({
        toBeWithinOf: function(distance, base) {
            this.message = function() {
                var lower = base - distance;
                var upper = base + distance;
                return "Expected " + this.actual + " to be between " +
                lower + " and " + upper + " (inclusive)";
            };
            return Math.abs(this.actual - base) <= distance;
        }
    });
});
```