Unit Testing

Dave van Soest

July 30, 2014

Purpose

Isolate parts of the program and prove that they work correctly.

Benefits

1. Find problems early/faster.
2. Facilitate change. (Refactoring!)
3. Simplify integration. (Bottom-up testing, ...)
4. Technical documentation. (Example uses, behavior.)
5. Improved design.
   - It's hard to write unit tests for a wrong design.
   - It's easy to write unit tests for a good design.
   - Testing units in isolation helps to expose tight coupling.
   - Separation of concerns, single-responsibility principle is promoted.

What is a unit?

Unit

A unit is a collection of functions and variables that is self-contained.

- A unit encompasses all functionality working on a certain piece of state, and all state needed by that functionality. (I.e.: it is self-contained.)
- So, no dependency on outside state!
- A unit needs proper design to eliminate direct dependencies and allow for proper unit testing.
  - For example, use the SOLID design principles as guidelines.

Candidate units:

- Functions
- Classes
- Namespaces
- Modules

Example

Find the unit to test...

```
1  var currentPos = {x: 100, y: 200};
2
3  var getCurrentPos = function() { return {x: currentPos.x, y: currentPos.y}; };
4  var setCurrentPos = function(x, y) { currentPos.x = x; currentPos.y = y; };
5
6  var updatePosition = function(velocity, timeDelta) {
7      var pos = getCurrentPos();
8      pos.x += velocity.x * timeDelta;
9      pos.y += velocity.y * timeDelta;
10     setCurrentPos(pos.x, pos.y);
11 };
```


Example

Refactored to support unit testing...

```
1  var currentPos = {x: 100, y: 200};
2
3  var getCurrentPos = function() { return {x: currentPos.x, y: currentPos.y}; };
4  var setCurrentPos = function(x, y) { currentPos.x = x; currentPos.y = y; };
5
6  var updatePosition = function(velocity, timeDelta) {
7      var pos = getCurrentPos();
8      pos = computeNewPosition(pos, velocity, timeDelta);
9      setCurrentPos(pos.x, pos.y);
10 };
11
12 var computeNewPosition = function(originalPos, velocity, timeDelta) {
13     var pos = {x: originalPos.x, y: originalPos.y};
14     pos.x += velocity.x * timeDelta;
15     pos.y += velocity.y * timeDelta;
16     return pos;
17 };
```

Skipped topics

1. Test suites.
2. Fixtures. Setup, teardown.
3. Assertions.
4. Testing frameworks.
5. Test result visualization.

Abstracting external depencies

Abstract away everything that uses...

- I/O (filesystem, network, ...)
- interprocess communication
- timers (time-outs, intervals)
- runtime environment
- undeterministic (random) behavior

How to abstract external depencies?

- Use dependency injection.
- Define an interface for the dependency.
- Create a fake or mock implementation of the interface.

What is faking, mocking and spying?

Faking

A fake provides an empty (stubbed) implementation of an interface, returning predefined results.

Mocking

A mock mimics functionality of a concrete implementation of an interface. It can potentially contain assertions.

Spying

A spy is a wrapper around a function, capturing information about the function's invocations. E.g., invocation count and call arguments. It can assert correct behavior of the caller.

When to use faking, mocking and spying?

- Fakes, mocks and spies are power tools. Use them judiciously. Prefer not to use them.
- Prefer faking or mocking complete interfaces.
- Only fake or mock methods of self-managed instances.
- Only fake or mock public interfaces.
- Use spying only for...
  - events
  - callback functions
  - injected self-managed instances

Example

Unit has direct dependency on I/O functionality, yikes!

```
1  var fs = require("fs");
2
3  function Person(name) {
4      this.name = name;
5  }
6
7  Person.prototype.load = function() {
8      var filePath = this.name + ".json";
9      var jsonData = fs.readFile(filePath);
10     var data = JSON.parse(jsonData);
11     this.name = data.name;
12 };
13
14 Person.prototype.save = function() {
15     var filePath = this.name + ".json";
16     var jsonData = JSON.stringify({name: this.name});
17     fs.writeFile(filePath, jsonData);
18 };
```


Example

Now using dependency injection...

```
1  function Person(name, fsImpl) {
2      this._fsImpl = fsImpl;
3      this.name = name;
4  }
5
6  Person.prototype.load = function() {
7      var filePath = this.name + ".json";
8      var jsonData = this._fsImpl.readFile(filePath);
9      var data = JSON.parse(jsonData);
10     this.name = data.name;
11 };
12
13 Person.prototype.save = function() {
14     var filePath = this.name + ".json";
15     var jsonData = JSON.stringify({name: this.name});
16     this._fsImpl.writeFile(filePath, jsonData);
17 };
```


Example

Even better, separated fs into multiple disjoint interfaces...

```
1  function Person(name) {
2      this.name = name;
3  }
4
5  Person.prototype.load = function(fileReaderImpl) {
6      var filePath = this.name + ".json";
7      var jsonData = fileReaderImpl.readFile(filePath);
8      var data = JSON.parse(jsonData);
9      this.name = data.name;
10 };
11
12 Person.prototype.save = function(fileWriterImpl) {
13     var filePath = this.name + ".json";
14     var jsonData = JSON.stringify({name: this.name});
15     fileWriterImpl.writeFile(filePath, jsonData);
16 };
```

Example

And this is the unit test...

```
function testSuite_Person() {
    function FileWriterMock(storage) {
        this._storage = storage;
    }
    FileWriterMock.prototype.writeFile = function(filePath, data) {
        this._storage[filePath] = data;
    };

    function FileReaderMock(storage) {
        this._storage = storage;
    }
    FileReaderMock.prototype.readFile = function(filePath) {
        return this._storage[filePath];
    };

    function testCase_saving_and_loading() {
        var storage = {};

        var person = new Person("Dave");
        var fileWriter = new FileWriterMock(storage);
        person.save(fileWriter);

        var fileReader = new FileReaderMock(storage);
        var samePerson = new Person();
        samePerson.load(fileReader);

        assert.equal(samePerson.name, person.name, "Names should be equal!");
    }
}
```


What to test? And what not?

What to test

- Public interface, public behavior.
- External events. (They are part of the unit's contract.)
- Extension points, in case the unit is a base class.

What not to test

- Non-public interface, non-public behavior.
  - Private methods.
  - Protected methods.
  - Internally scoped units.
- Effects on non-public properties.

Why only test public interfaces?

- Goal of the test is to prove that the unit adheres to its public contract.
- The rest of the program (if properly designed) does not depend on the inner workings of the unit.
- Non-public functions/methods are not obliged to leave the unit in a valid state as a post-condition.
- As long as the unit's public interface doesn't change, its internals are allowed to be turned 180 degrees around.
- See unit testing benefits number 2, 4 and 5.

Why only test public interfaces?

(continued...)

- Non-public functions can not be treated as units. They rely on internals outside themselves, which might be subject to change.
  - If a non-public function can be treated as a unit, there is no reason for it to be non-public.
- A unit with complicated private functions probably has a hidden implicit internal class wanting to get out and which should be independently tested. The public functions are likely just facades.

The Refactoring Experiment

One of the big benefits of unit testing is being able to confidently refactor the implementation of a unit.

It should always be possible to arbitrarily refactor any internals of a unit, so that without modifying any of its unit tests, these tests still run successfully.

If not, then...

- the unit tests are written against non-public parts of the interface,
- or the unit being tested is not an actual unit, i.e. it is not self-contained.

Example

Unit being tested...

```
1  function Player(pos) {
2      this._pos = {x: pos.x, y: pos.y};
3  }
4  Pet.prototype.setX = function(x) { this._pos.x = x; };
5  Pet.prototype.getX = function() { return this._getPosCoord("x"); };
6  Pet.prototype._getPosCoord = function(coord) { return this._pos[coord]; };
```

Unit test using non-public interface... (Wrong! But succeed.)

```
1  function testSuite_Player() {
2      function testCase_construct() {
3          var p = new Player({x: 10, y: 20});
4          assert.equal(p._getPosCoord("x"), 10, "X␣pos.␣should␣be␣set␣by␣ctor.");
5      }
6      function testCase_setX() {
7          var p = new Player({x: 0, y: 0});
8          p.setX(30);
9          assert.equal(p._pos.x, 30, "X␣pos.␣should␣be␣set␣by␣setter.");
10     }
11     function testCase_getX() {
12         var p = new Player({x: 0, y: 0});
13         p._pos = {x: 50, y: 10};
14         assert.equal(p.getX(), 50, "X␣pos.␣should␣be␣retrieved␣by␣getter.");
15     }
16 }
```


Example

Unit after refactoring...

```
1  function Player(pos) {
2      this._pos = new Vector([pos.x, pos.y]);
3  }
4  Pet.prototype.setX = function(x) { this._pos.set(0, x); };
5  Pet.prototype.getX = function() { return this._pos.get(0); };
```

Original tests... (Fail!)

```
1  function testSuite_Player() {
2      function testCase_construct() {
3          var p = new Player({x: 10, y: 20});
4          assert.equal(p._getPosCoord("x"), 10, "X␣pos.␣should␣be␣set␣by␣ctor.");
5      }
6      function testCase_setX() {
7          var p = new Player({x: 0, y: 0});
8          p.setX(30);
9          assert.equal(p._pos.x, 30, "X␣pos.␣should␣be␣set␣by␣setter.");
10     }
11     function testCase_getX() {
12         var p = new Player({x: 0, y: 0});
13         p._pos = {x: 50, y: 10};
14         assert.equal(p.getX(), 50, "X␣pos.␣should␣be␣retrieved␣by␣getter.");
15     }
16 }
```


Example

Correct unit tests on public interface, support refactoring...

```
1  function testSuite_Player() {
2      function testCase_get_and_set() {
3          var p = new Player({x: 10, y: 20});
4          assert.equal(p.getX(), 10, "X␣pos.␣should␣be␣set␣by␣ctor.");
5          assert.equal(p.getY(), 20, "Y␣pos.␣should␣be␣set␣by␣ctor.");
6          p.setX(30);
7          assert.equal(p.getX(), 30, "X␣pos.␣from␣get␣should␣be␣set␣by␣setter.");
8      }
9  }
```

Red flags, part 1

Signs that something is wrong and action is required

- It's hard to write unit tests for a unit.
  - Unit design issue.
- Mocking/faking/spying of functions of the unit being tested.
  - The test subject is modified to prove it works. Something's definitely wrong.
  - Unit design issue or error in unit test.
- Non-public interface is being tested.
  - Probable cause: There is a separate unit hidden inside the unit being tested.
- Unit depends on state outside of itself.
  - Either refactor the unit using dependency injection,
  - Or choose the bigger unit to write the unit tests against.
- Inspecting the value of non-public properties.
  - Don't!
  - Just don't!

Red flags, part 2

Signs that something is wrong and action is required

- Mocking/Faking instances not managed by the testing code.
  - Create abstraction of the instance class using interfaces.
- Unit functions are tested individually, instead of combined.
  - In case the functions don't operate on shared state: refactor the unit by splitting it up in smaller units.
  - Otherwise: probably non-public properties are inspected.
  - Write unit tests that test/demonstrate the behavior of (sequentially) invoking a combination of the unit's functions. Because this public behavior is the unit's contract with its clients and therefore should be tested.
- Changes in the unit source code are needed to support unit testing.
  - Don't!
  - Probably non-public behavior is being tested.

Questions & Discussion