Unit-Tests für JavaScript und AJAX

Peter Krenn

10. Januar 2011

Zusammenfassung

Durch den Einsatz von JavaScript in komplexer werdenden Web- und Server-Anwendungen bei einer größer werdenden Anzahl von JavaScript Implementierungen gewinnt JavaScript Testing an Bedeutung. Dieser Artikel stellt aktuelle Methoden zu Unit Testing, Behaviour Driven Development, Mocking/Stubbing und Regression Testing mit JavaScript vor.

Inhaltsverzeichnis

1 Einleitung

Laut einer Umfrage[28] testen nur 34% der Entwickler ihren JavaScript Code. Im Vergleich zur vorjährigen Version der Umfrage[27] gab es nur eine Verbesserung um 2%. Trotzdem existiert eine Vielzahl von JavaScript Testing Frameworks.

Die verbreitetsten Open Source JavaScript Frameworks und Libraries, wie zum Beispiel jQuery[19], Prototype/Scriptaculous[12], MooTools[15] oder YUI[7] verfügen alle über umfassende Test Suites.

Die von den JavaScript Frameworks verwendeten Test Frameworks sind auch die am meisten eingesetzten: QUnit[29] (jQuery) 34%, Jasmine[9] (MooTools) 18%, YUITest[8] (YUI) 6%, unittest.js[3] (Prototype) 5%.

Verbreitet ist weiterhin JsUnit[13] mit 12%. Dieses Framework war das erste JavaScript Unit Testing Framework und die Entwicklung daran begann 2001, wurde aber mittlerweile eingestellt.

Neben Browser-basiertem Testen gewinnt Server-seitiges Testen an Bedeutung, bereits 35% der Entwickler verwenden JavaScript in einer Server Umgebung. Node.js beinhaltet eine Implementierung der CommonJS Assertion Testing Spezifikation[16].

Ebenfalls für Server-seitiges Testen relevant sind Headless Testing Frameworks, wie zum Beispiel Jasmine. Sie sind nicht vom DOM-Modell des Browsers oder anderen Frameworks abhängig und können auch von der Kommandozeile aus aufgerufen werden.

Auch Mocking und Stubbing Libraries wurden für JavaScript entwickelt. Zum Beispiel Sinon.JS[11] funktioniert mit jedem Unit Testing Framework und erlaubt es verschiedene Arten von Test Doubles und Fake XHR Requests zu erzeugen.

Der weitere Artikel wird sich vor Allem mit der Anwendung von QUnit und Jasmine beschäftigen. Diese beiden Test Frameworks sind die, an denen zur Zeit am aktivsten entwickelt wird.

2 QUnit

QUnit[29] wird von Mitgliedern des jQuery Teams[19] entwickelt und ist auch die offizielle *test suite* des jQuery Projektes. Es zählt zur Gruppe der xUnit Test Frameworks[2], die einem Design folgen, das erstmals mit SUnit[1] für Smalltalk formuliert wurde und als JUnit[4] für Java große Verbreitung fand.

2.1 Assertions

Der zentrale Bestandteil eines *unit tests* ist die Assertion. Eine Assertion ist eine Aussage über den Zustand einer bestimmten Stelle in einem Programm[23]. Der Entwickler macht eine Zusicherung, dass eine bestimmte Bedingung unabhänging von den Laufzeitumständen immer wahr ist.

QUnit stellt eine Reihe von allgemeinen Assertions zur Verfügung, die JUnit nachempfunden sind, und auch solche, die speziell an JavaScript und asynchrone Entwicklung angepasst sind[29]:

`ok(state, message)`

`ok()` ist eine boolsche Assertion, die positiv terminiert, wenn der erste Parameter wahr ist. Den zweiten, optionalen Parameter `message` haben alle Assertion-Funktionen gemein: dieser wird gemeinsam mit dem Ergebnis ausgegeben.

```
ok(true, "terminiert immer positiv");
ok("", "Leerstring entspricht false, die assertion schlägt fehl");
```

`equal(actual, expecteded, message)`

Diese Assertion ist `ok()` sehr ähnlich, allerdings werden die `actual` und `expecteded` Werte mit dem Ergebnis ausgegeben. Dadurch werden die Tests aussagekräftiger und Debugging einfacher. `equal()` terminiert positiv, wenn `actual == expected`.

```
var actual = 1;
equal(actual + 1, 2, "terminiert positiv");
equal(actual, 2, "schlägt fehl");
```

`strictEqual(actual, expected, message)`

Im Gegensatz zu `equal()` werden bei dieser Assertion die Werte mit `actual === expected` verglichen. Dieser Operator vergleicht zusätzlich die Typengleichheit der beiden Parameter.

```
var actual = 0;

// terminiert positiv, da == den Datentyp ignoriert
equal(actual, false);

// schlägt fehl, da 0 vom Typ Number und false Boolean ist
strictEqual(actual, false);
```

deepEqual(actual, expected, message)

`deepEqual()` überprüft primitive Datentypen, Arrays und Objekte rekursiv auf Gleichheit. Sie ist dabei auch strikter als `equal()`, da der === Operator verwendet wird.

```
var actual = {a: 1};

// equal schlägt mit verschiedenen Objekten fehl
equal(actual, {a: 1});

// schlägt fehl, da die Datentypen verschieden sind;
deepEqual(actual, {a: "1"});

// terminiert positiv, da die Inhalte der Objekte gleich sind
deepEqual(actual, {a: 1}
```

notEqual(actual, expected, message)

Diese Assertion ist die invertierte Version von `equal()`. Entsprechende Funktionen existieren auch für `strictEqual()` und `deepEqual()`: `notStrictEqual()` und `notDeepEqual`.

```
var actual = 0;
notEqual(actual, false, "schlägt fehl");
notStrictEqual(actual, false, "terminiert positiv");

var actual = {a: 1};
notDeepEqual(actual, {a: "1"}, "terminiert positiv");
```

raises(state, message)

`raises()` überprüft, ob die übergebene Callback Funktion eine Exception wirft. Die Funktion wird ohne Parameter im *default scope* aufgerufen.

```
raises(function() {
  throw new Error("Fehlerfall");
}, "terminiert positiv, falls der gewünschte Fehler auftritt");
```

2.2 Testfunktionen

Werden die vorgestellten Assertions direkt ausgeführt, unterbrechen sie im Falle eines Fehlers den Testdurchlauf. Deswegen werden sie in Testfunktionen einge-

bettet.

Es ist auch üblich, dass jede Testfunktion nur eine spezifische Eigenschaft testet. Dies geschieht oft mit mehreren Assertions und die Testfälle bleiben dabei überschaubar und einfach zu verstehen.

QUnit macht das mit der Funktion `test(name, expected, test)`. `name` ist eine Bezeichnung des Tests und `test` ist eine Funktion, die die Assertions enthält. `expect` ist optional und erlaubt es die Anzahl der zu erwartenden Assertions festzulegen. Dies ist bei asynchronen Tests von Bedeutung.

```
Beispiel
```

2.3 Test Suites

Mit der `module(name, lifecycle)` Anweisung ist es möglich, die Testfälle weiter in Module aufzuteilen. Dies dient einerseits der Strukturierung der Tests und andererseits wird es auch möglich, für mehrere Testfälle relevante Daten vor deren Ausführung zu definieren.

```
Beispiel
```

Im Parameter `lifecycle` kann man optional die Funktionen `setup()` und `tearDown()` übergeben, die vor beziehungsweise nach jedem Test ausgeführt werden. Sie teilen den `this`-*scope* der Testfunktionen, wodurch sogenannte *fixtures* erstellt werden können. Die Daten werden nach jedem Test zurückgesetzt, was in `tearDown()` auch manuell geschehen kann.

2.4 Asynchrones Testen

Asynchrone Funktionen, wie sie bei AJAX Requests und Aufrufen von `setTimeout()` und `setInterval()` vorkommen, können mit den bisher vorgestellten Methoden nicht getestet werden.

Im folgenden Beispiel[6] wird die Assertion nicht ausgeführt, da das Ergebnis des Testfalles zum Zeitpunkt des Aufrufens bereits feststeht.

```
test("Asynchroner Test", function() {
  setTimeout(function() {
    ok(true);
  }, 100)
});
```

QUnit stellt aus diesem Grund zwei Funktionen zur Verfügung, mit der sich der Testablauf pausieren und fortsetzen lässt.

```
test("Asynchroner Test", function() {
  stop();

  setTimeout(function() {
    ok(true);

    start();
  }, 100)
})
```

`stop()` hält den Test an und nach Aufrufen der Assertion, wird mit `start()` wieder fortgesetzt. Da der Fall, dass `stop()` am Anfang eines Tests ausgeführt wird, häufig vorkommt, gibt es dafür auch ein spezielle Funktion namens `asyncTest()`.

Wird nicht ein `setTimeout()` Aufruf sondern ein AJAX Request getestet, bei dem man sich nicht sicher sein kann, wann und ob die *callback* Funktion ausgeführt wird, ist es möglich der `stop()` Funktion ein Intervall als Parameter mitzugeben, nachdem der Testablauf automatisch fortgesetzt wird. Der betreffende Test schlägt dabei fehl.

Werden in einem Testfall mehrere zeitverzögerte Aufrufe getestet, ist es notwendig, `start()` manuell mit `setTimout()` ausreichend zu verzögern, so dass alle Callbacks ausreichend Zeit zum Terminieren haben.

Um sicher zu stellen, dass alle Callbacks mit den enthaltenen Assertions ausgeführt werden, gibt es eine `expect()` Funktion, der man als Parameter die zu erwartende Anzahl der Assertions übergeben kann.

```
// Ein AJAX request mit der zu testenden callback Funktion
function ajaxCall(successCallback) {
  jQuery.ajax({
    url: "http://server.com/",
    success: successCallback
  });
}

asyncTest("Asynchroner Test", function() {
  // asyncTest unterbricht den Testablauf

  // 3 assertions sollten ausgeführt werden
  expect(3);

  ajaxCall(function() {
    ok(true);
  });

  ajax(function() {
    ok(true);
    ok(true);
  })

  // Nach 3000 ms wird der Testablauf fortgesetzt
  setTimeout(function() {
    start();
  }, 3000);
});
```

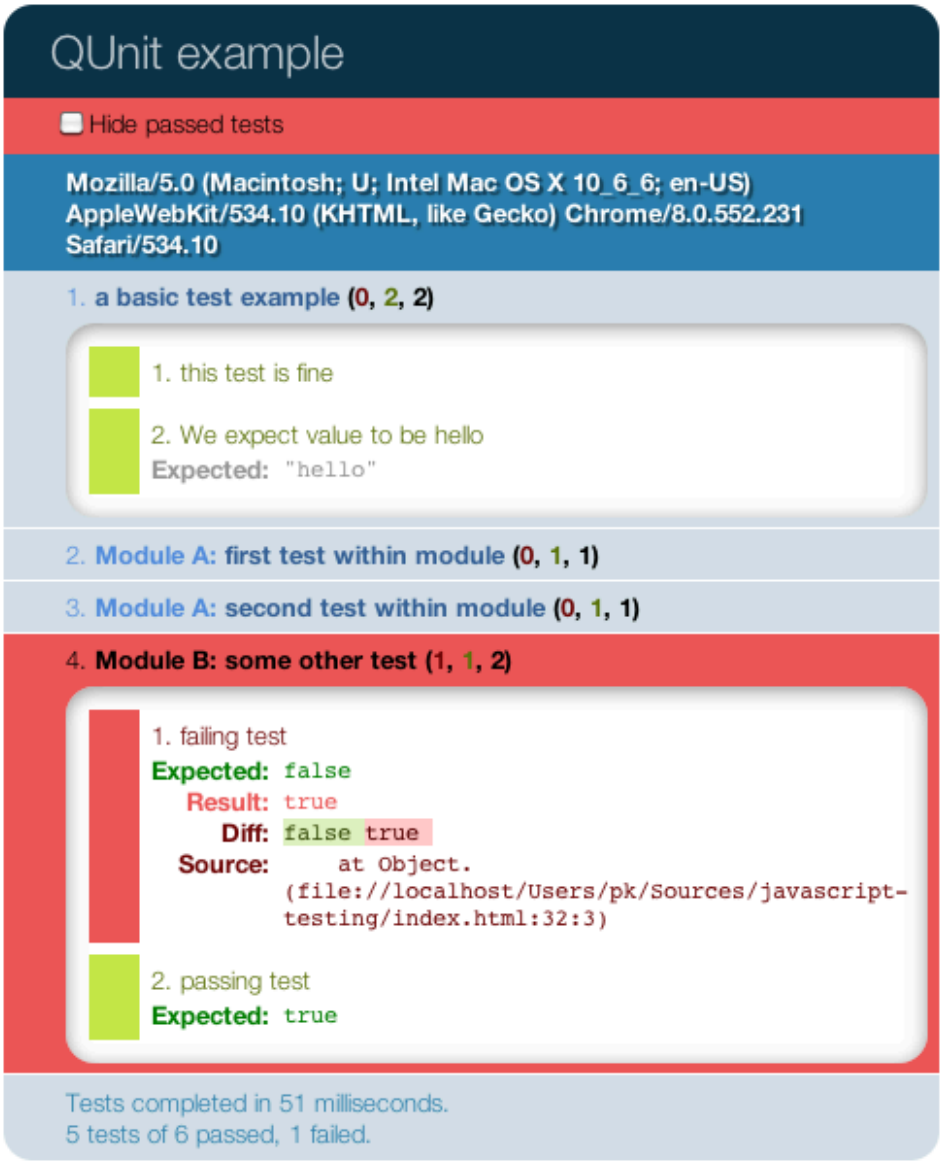
Die Kombination von `expect()` mit `asyncTest()` ist wiederum so verbreitet, dass die Anzahl der zu erwartenden Assertions als zweiter Parameter direkt in den `asyncTest()` Aufruf verlegt werden kann.

2.5 Testablauf

Die Ausgabe von QUnit erfolgt ausschließlich im Browser. Die Dateien `qunit.js` und `qunit.css` müssen gemeinsam mit dem zu testenden JavaScript Code und den Tests in eine HTML-Datei eingebunden werden.

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <script type="text/javascript" src="qunit.js"></script>
    <link rel="stylesheet" href="qunit.css" type="text/css"
          media="screen" />

    <!-- Projekt Datei -->
    <script type="text/javascript" src="my_project.js"></script>

    <!-- Datei mit den Tests -->
    <script type="text/javascript" src="my_tests.js"></script>
  </head>
  <body>
    <h1 id="qunit-header">QUnit example</h1>
    <h2 id="qunit-banner"></h2>
    <div id="qunit-testrunner-toolbar"></div>
    <h2 id="qunit-userAgent"></h2>
    <ol id="qunit-tests"></ol>
    <div id="qunit-fixture">test markup, will be hidden</div>
  </body>
</html>
```

Beim Ausführen der Datei im Browser befüllt QUnit die Elemente im `body` mit den Ergebnissen der Tests.



Für jeden Testfall wird die Anzahl der erfolgreichen und fehlgeschlagenen Assertions und auch deren Gesamtanzahl angegeben. Für fehlgeschlagene Tests wird dessen Position im Quellcode angezeigt.

QUnit ist nur für die Verwendung im Browser ausgelegt, ein Kommandozeilen-basierter Aufruf der Testfälle ist nicht vorgesehen. Ein Versuch[17] QUnit in der simulierten Browserumgebung Env.js[18] anzuwenden, war nur begrenzt erfolgreich, da der Testbericht im HTML-Format ausgegeben wird. Eine Alternative ist das weiter unten beschriebene TestSwarm Projekt.

3 Jasmine

Jasmine[9] gehört zur Gruppe der Behavior Driven Development[24] oder BDD Frameworks. Im Gegensatz zu traditionellen xUnit Test Frameworks versuchen diese die Sprache, in der die Tests verfasst sind, einfacher und damit auch für nicht-technische Projektteilnehmer verständlich zu machen. BDD soll dabei helfen, die gewünschte *software behavior* in Diskussion mit den Auftraggebern klar zu definieren.

Im Gegensatz zu seinem Vorgänger Screw.Unit[22] und QUnit ist Jasmine ein Headless Testing Framework, das heißt es benötigt keinen Browser und ist nicht von anderen JavaScript Libraries abhängig. Dies vereinfacht den automatisierten Aufruf der Tests und ermöglicht es Server-side Anwendungen zu testen. Wie QUnit unterstützt es asynchrones Testen und verfügt außerdem über Mocking und Stubbing Funktionalitäten.

3.1 Specs, Expectations und Suites

Specs entsprechen den Testfunktionen von klassischen xUnit Test Frameworks. Sie beschreiben das Verhalten einer Funktion oder eines Objektes mit einem Aufruf der Funktion `it()` in Form einer natürlichen Sprache. Die Beschreibung sollte aussagekräftig sein, da sie im Testbericht ausgegeben wird.

Expectations drücken die Erwartungen an das Verhalten eines bestimmten Teils des Quellcodes mittels der `expect()` Funktion aus. Sie sind vergleichbar mit den Assertions der xUnit Frameworks. Das gewünschte Verhalten wird durch Matchers wie `toEqual` spezifiziert:

```
it("should increment a variable", function() {
  // Definition der Testdaten
  var value = 0;

  // Ausführen des zu testenden codes
  value++;

  // Testen der Erwartung
  expect(value).toEqual(1);
});
```

Zur Strukturierung der Specs dienen Suites. Wie bei QUnit ist es möglich, Anweisungen vor und nach dem Aufrufen jedes Tests auszuführen – dies geschieht mit `beforeEach()` und `afterEach()`.

```
describe("Calculator", function() {
  var a, b;

  beforeEach(function() {
    a = 3;
    b = 4;
  });

  it("should add numbers with add()", function() {
    expect(Calculator.add(a, b)).toEqual(7);
  });

  it("should multiply numbers with multiply()", function() {
    expect(Calculator.multiply(a, b)).toEqual(12);
  });
});
```

3.2 Matchers

Jasmine verfügt über eine Reihe von eingebauten Matchers und deren Inversionen, die man durch Voranstellen von `.not` erhält:

`toEqual()`

Dieser Matcher entspricht der QUnit Assertion `deepEqual()` und kann außerdem Datumswerte und DOM-Nodes vergleichen.

```
// Alle expectations terminieren positiv
expect(true).toEqual(true);
expect(false).not.toEqual(true);

expect("5").not.toEqual(5);

expect({a: 1}).not.toEqual({a: "1"});

expect(new Number("5")).toEqual(5);

expect(new Date("January 11, 2011, 12:00:00"))
  .not.toEqual(new Date("January 12, 2011, 12:30:00"));

var nodeA = document.createElement("div");
var nodeB = document.createElement("div");
expect(nodeA).toEqual(nodeA);
expect(nodeA).not.toEqual(nodeB);
```

toBe()

`toBe()` entspricht `strictEqual()`.

```
var a = {};
var b = {};
var c = b;

expect(a).not.toBe(b);
expect(c).toBe(b);
```

toMatch()

`toMatch()` überprüft Strings auf Übereinstimmungen mit Regulären Ausdrücken.

```
expect("javascript").toMatch(/script/);
expect("javascript").not.toMatch(/testing/);
```

toBeDefined()

`toBeDefined()` terminiert positiv für definierte Werte.

```
var a = 5;

expect(a).toBeDefined();

expect(undefined).not.toBeDefined();
```

`toBeNull()`

`toBeNull()` erwartet einen `null`-Wert.

```
expect(null).toBeNull();
expect(undefined).not.toBeNull();
```

`toBeTruthy()` und `toBeFalsy()`

Diese Matchers überprüfen, ob ein Ausdruck in `true` beziehungsweise `false` evaluiert.

```
expect(0).not.toBeTruthy();
expect("").toBeFalsy();
expect(5).toBeTruthy();
expect({a: 1}).not.toBeFalsy();
```

`toContain()`

`toContain()` terminiert positiv, falls der übergebene String oder Array ein bestimmtes Zeichen oder Element enthält.

```
expect("javascript").toContain("a");
expect("javascript").not.toContain("x");

expect([1, 2, 3]).toContain(3);
expect([1, 2, 3]).not.toContain("x");
```

`toBeLessThan()` und `toBeGreaterThan()`

Diese Matchers überprüfen, ob ein Wert kleiner beziehungsweise größer als der Übergebene ist.

```
expect(5).toBeLessThan(10);
expect(5).not.toBeGreaterThan(10);
```

`toThrow()`

Mit `toThrow()` drückt man die Erwartung aus, dass ein festgelegter Ausdruck die übergebene Exception wirft.

```
expect(function() { throw new Error("error"); }).toThrow("error");
```

Benutzerdefinierte Matchers

Um die Lesbarkeit der Specs zu verbessern, ist es mit Jasmine auch möglich eigene Matchers zu definieren. Eine Matcher Funktion erhält den zu vergleichenden Wert als `this.actual` und es können zusätzlich beliebig viele Parameter übergeben werden. Die benutzerdefinierten Matchers werden mit `this.addMatchers()` in der `beforeEach()` oder der `it()` Funktion definiert.

```
beforeEach(function() {
  this.addMatchers({
    toBeLessOrEqualThan: function(expected) {
      return this.actual <= expected;
    }
  });
});
```

3.3 Asynchrones Testen

Zum Testen von asynchronen Funktionen stellt Jasmine die Funktionen `runs()`, `waits()` und `waitsFor()` bereit.

Von `runs()` aufgerufene Funktionen für sich alleine verhalten sich genauso, als wären sie direkt aufgerufen worden. In Verbindung mit `waits()` kann der Testablauf für eine bestimmte Zeit unterbrochen werden, bis die nächste `runs()` Funktion aufgerufen wird. Die `runs()` Blöcke innerhalb eines `it()` Blocks teilen den gleichen `this`-*scope*.

```
it("should increment a value after 250 ms", function() {
  runs(function() {
    this.value = 0;

    // 250 ms in der Zukunft wird value erhöht
    setTimeout(function() {
      this.value++;
    }, 250);
  });

  // value entspricht noch dem initialisierten Wert
  runs(function() {
    expect(this.value).toEqual(0);
  });

  waits(500);

  // 500 ms später wurde value korrekt erhöht
  runs(function() {
    expect(this.value).toEqual(1);
  });
});
```

`waitsFor()` pausiert den Testablauf, bis ein bestimmtes Ereignis eintrifft. Als erster Parameter wird eine Funktion übergeben, die `true` im Falle des Eintreffens des Ereignisses zurück gibt. Optional kann auch die maximale Wartedauer und eine Fehlermeldung übergeben werden.

```
describe('Calculation', function() {
  it('should calculate a value asynchronously', function() {
    var calculation = new Calculation();
    calculation.asynchronouslyCalculateValue();

    // waitsFor() wartet bis isComplete() true zurück gibt
    // maximal aber eine Sekunde
    waitsFor(function() {
      return calculation.isComplete();
    }, "calculation wasn't completed", 1000);

    runs(function() {
      expect(calculation.value).toEqual(1);
    });
  });
});
```

3.4 Spies

Jasmine Spies sind Test Doubles[14], die als Stubs, Spies oder in Verbindung mit einer Expectation als Mocks agieren können. Test Doubles sind Objekte, die das Interface oder Verhalten eines realen Objektes simulieren und werden eingesetzt um die Komplexität der Tests zu reduzieren. Bei Projekten, die viele verschachtelte Objekte und Module umfassen, macht es Sinn, sich beim Testen nur auf relevante Teile zu konzentrieren und bei Kommunikation nur auf die korrekte Verwendung der Interfaces zu achten. Für JavaScript relevante Beispiele sind asynchrone Aufrufe oder AJAX Requests.

Test Stubs sind Objekte, die ausschließlich auf die für den Test notwendigen Aufrufe reagieren. Eine erweiterte Version sind Test Spies, sie zeichnen die Anzahl und Parameter der Aufrufe auf, welche später in Assertions oder Expectations getestet werden können. Mocks sind vordefinierte Objekte mit eingebauten Expectations, die bestimmen welche Aufrufe erwartet werden.

`spyOn(object, methodName)` verhindert im ersten Schritt den Aufruf einer Funktion und zeichnet folgende Eigenschaften auf: `callCount` enthält wie oft die Funktion aufgerufen wurde, `mostRecentCall.args` ist Parameter-Array des letzten Aufrufes und mit `argsForCall[]` kann man auf die Parameter sämtlicher Aufrufe zugreifen.

```
// Ein Objekt, dessen Interface simuliert wird
var example = {
  not: function(boolean) { return !boolean; }
};

// Ein Spy wird für example.not angelegt
spyOn(example, "not");
example.not(true);

// example.not.callCount: 1
// example.mostRecentCall.args: [true]
```

Im Falle eines Aufrufes kann die Reaktion eines Spy festgelegt werden:

```
// ruft die originale Funktion auf
spyOn(example, "not").andCallThrough();

// gibt die übergebene Werte bei Aufruf zurück
spyOn(example, "not").andReturn(true);

// wirft bei Aufruf eine Exception
spyOn(example, "not").andThrow("error");

// ruft die übergebene Funktion auf
spyOn(example, "not").andCallFake(function() { return true; });
```

Mit folgenden Matchers können die Spies in Expectations verwendet werden. Dadurch verhalten sich die Funktionen als Mocks.

```
spyOn(example, "not");
example.not(true);

expect(example.not).toHaveBeenCalled();
expect(example.not).not.toHaveBeenCalledWith(false);
```

Anstatt eine bestehende Funktion zu einem Spy zu machen, ist es auch manchmal sinnvoll, mit `jasmine.createSpy()` ein Stub-artiges Objekt manuell zu erstellen. Ein Anwendungsgebiet ist das Testen von Callbacks[10].

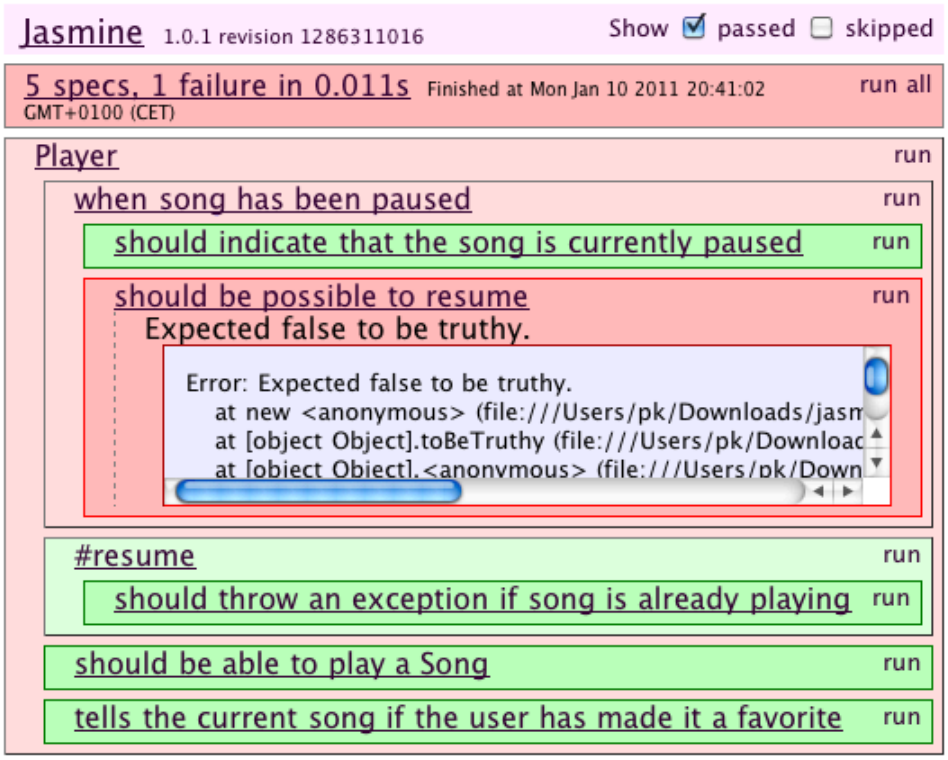
```
var Klass = function() {
};

Klass.prototype.methodWithCallback = function(callback) {
  return callback("example");
};

it("should spy on Klass#methodWithCallback", function() {
  var callback = jasmine.createSpy();
  new Klass().methodWithCallback(callback);

  expect(callback).toHaveBeenCalledWith("example");
});
```

Eine weitere Anwendung ist das Testen von AJAX Request oder asynchronen Funktionen, die Callbacks als Parameter verlangen. Die Request beziehungsweise die Funktion wird dabei nicht ausgeführt, sondern die Callback Funktion wird isoliert getestet.

```
var Klass.prototype.asyncMethod = function(callback) {
  asyncCall(callback);
};

it("should test async call") {
  spyOn(Klass, 'asyncMethod');
  var callback = jasmine.createSpy();

  Klass.asyncMethod(callback);
  expect(callback).not.toHaveBeenCalled();

  // Die an asyncMethod übergebene callback Funktion
  // wird manuell getestet
  Klass.asyncMethod.mostRecentCall.args[0]("example");
  expect(callback).toHaveBeenCalledWith("example");
});
```

3.5 Testablauf

Für Jasmine gibt es eine Stand-alone Version, deren Funktionalität der von QUnit entspricht. In einer HTML-Datei werden die Test-, sowie die Projektdateien eingebunden und beim Öffnen im Browser wird sie mit den Testergebnissen befüllt.



Nicht erfüllte Expectations werden farblich markiert und der Stack für den jeweiligen Fehler wird angezeigt. Zusätzlich ist es möglich, einzelne Specs vom Browser aus erneut aufzurufen.

Mit dem Kommandozeilen Interface von Jasmine ist es möglich automatisiert verschiedene Browser zu starten und die Test Suites auszuführen. Die Browser werden von Selenium[21] geladen und geben die Ergebnisse über die Kommandozeile aus.

Ein Beispiel für die Server-seitige Anwendung von Jasmine ist jasmine-node[5], eine Adaptierung für node.js. Die Specs können direkt mit dem node.js Interpreter ausgeführt und ausgegeben werden.

4 Regression Testing

Regression Testing[26] versucht Probleme und Nebenwirkungen von Modifikationen im Quellcode durch Wiederholung der Testfälle aufzuzeigen. Ein verwandter Begriff ist Continuous Integration[25], die kontinuierliche Qualitätskontrolle für ein Projekt erreichen will. Automatisiertes Testing ist eines ihrer Ziele.

Da es keine einheitliche JavaScript Implementierung gibt, sondern diese sich je

nach Browser, Browserversion und Betriebssystem unterscheiden, ist die Durchführung der Testläufe problematisch.

Ein Ansatz, der sich derzeit in Entwicklung befindet, ist TestSwarm[20] von John Resig, dem Autor von jQuery und QUnit. Es handelt sich dabei um ein System, das den Code eines Projektes bei jedem *commit* im Versionierungssystem automatisch auf verschiedenen Plattformen und Browsern testet.

Das System wird von einem zentralen Server aus gesteuert, der Testaufträge an mehrere Clients, auf denen verschiedene Browserversionen laufen, weiterleitet. Das Ergebnis der Prozesses ist eine tabellarische Ansicht die für jeden *commit* darstellt, auf welchen Plattformen und Browsern die Test Suite fehlerfrei angewendet worden ist.

Im Gegensatz zu vergleichbaren Systemen, wie Selenium[21], ist TestSwarm vom verwendeten Test Framework unabhängig: es werden alle im Moment gebräuchlichen Frameworks unterstützt.

Literatur

[1] Kent Beck. Simple smalltalk testing: With patterns. *Smalltalk Report*, 4(2):16–18, 1994.

[2] Martin Fowler. xUnit. http://www.martinfowler.com/bliki/Xunit.html, 2010.

[3] Thomas Fuchs. unittest.js. http://madrobby.github.com/scriptaculous/unit-testing/, 2011.

[4] Erich Gamma and Kent Beck. JUnit. http://junit.org/, 2011.

[5] Miško Hevery. jasmine-node. https://github.com/mhevery/jasmine-node, 2011.

[6] Gaving Huang. How to test your JavaScript code with QUnit | nettuts+. http://net.tutsplus.com/tutorials/javascript-ajax/how-to-test-your-javascript-code-with-qunit/, 2010.

[7] Yahoo Inc. YUI library. http://developer.yahoo.com/yui/, 2011.

[8] Yahoo Inc. YUI test. http://developer.yahoo.com/yui/yuitest/, 2011.

[9] Jasmine. Jasmine - BDD for javascript. http://pivotal.github.com/jasmine/, 2011.

[10] Jasmine. Jasmine - spies. http://pivotal.github.com/jasmine/spies.html, 2011.

[11] Christian Johansen. Sinon.JS - versatile standalone test spies, stubs and mocks for JavaScript. http://sinonjs.org/, 2011.

[12] Prototype JS. Prototype JavaScript famework. http://www.prototypejs.org/, 2011.

[13] JsUnit. JsUnit. http://www.jsunit.net/, 2011.

[14] Gerard Meszaros. Test double. http://xunitpatterns.com/Test%20Double.html, 2011.

[15] MooTools. MooTools - a compact JavaScript framework. http://mootools.net/, 2011.

[16] Node.js. Node.js - assertion testing. http://nodejs.org/docs/v0.3.4/api/assert.html, 2011.

[17] Benjamin Plee. QUnit & rhino. http://twoguysarguing.wordpress.com/2010/11/02/make-javascript-tests-part-of-your-build-qunit-rhino/, 2010.

[18] John Resig. Envjs. http://www.envjs.com/, 2011.

[19] John Resig. jQuery. http://jquery.com/, 2011.

[20] John Resig. TestSwarm. https://github.com/jeresig/testswarm/wiki, 2011.

[21] Selenium. Selenium web application testing system. http://seleniumhq.org/, 2011.

[22] Nathan Sobo. Screw.Unit. https://github.com/nathansobo/screw-unit, 2011.

[23] Wikipedia. Assertion (computing). http://en.wikipedia.org/wiki/Assertion_(computing), 2011.

[24] Wikipedia. Behavior driven development. http://en.wikipedia.org/wiki/Behavior_Driven_Development, 2011.

[25] Wikipedia. Continuous integration. http://en.wikipedia.org/wiki/Continuous_integration, 2011.

[26] Wikipedia. Regression testing. http://en.wikipedia.org/wiki/Regression_testing, 2011.

[27] Alex Young. DailyJS - JavaScript developer survey results 2009. http://dailyjs.com/2009/12/02/survey-results/, 2009.

[28] Alex Young. DailyJS - JavaScript developer survey results 2010. http://dailyjs.com/2010/12/13/javascript-survey-results/, 2010.

[29] Jörg Zaefferer and John Resig. QUnit. http://docs.jquery.com/Qunit, 2011.