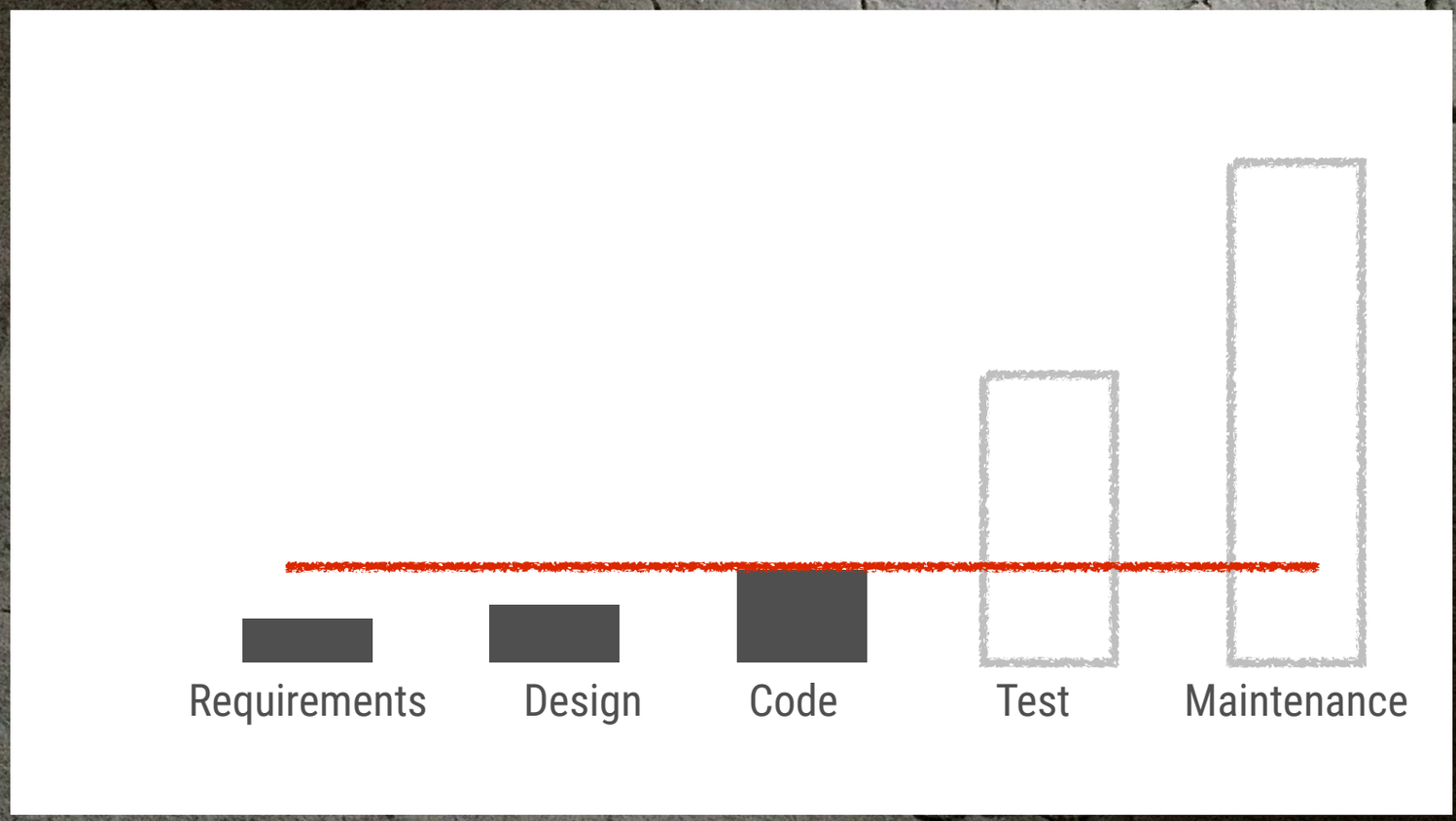
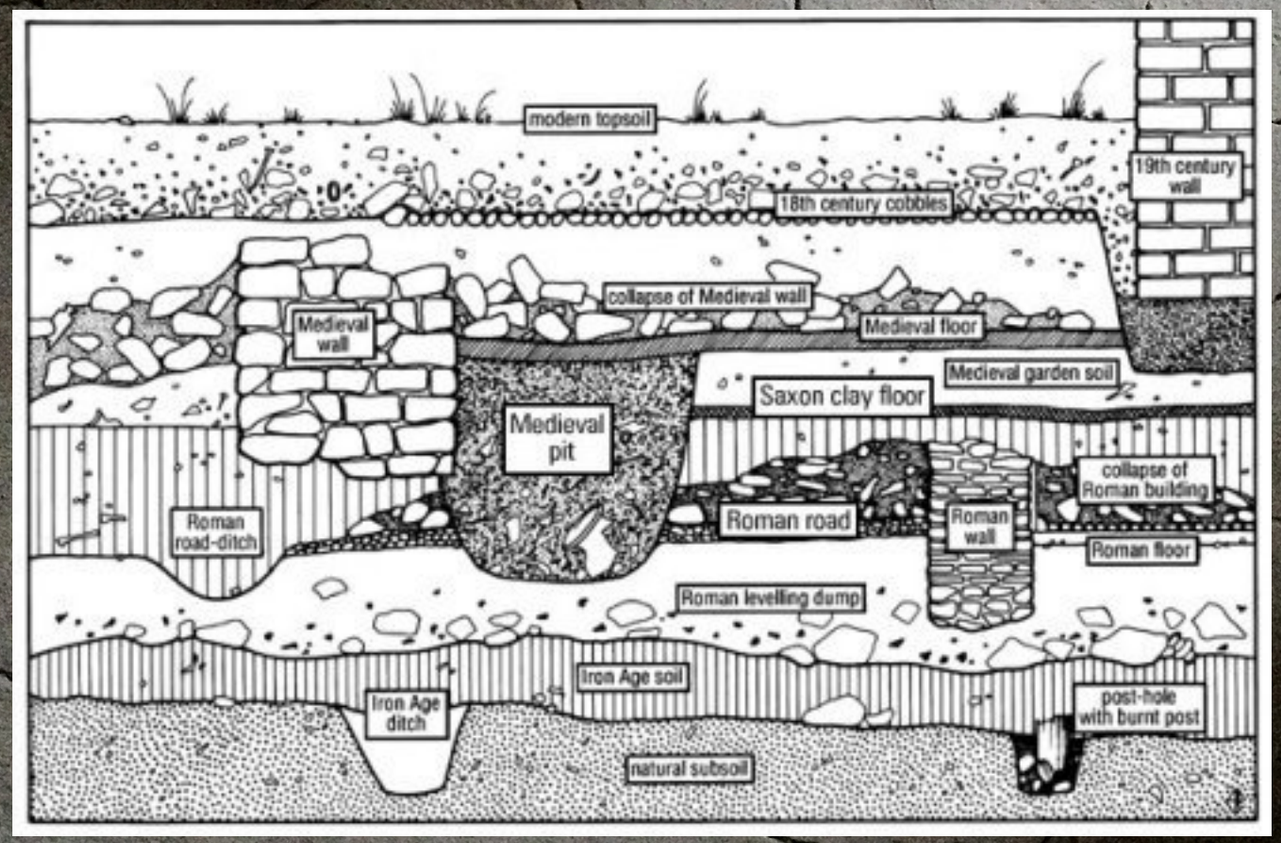
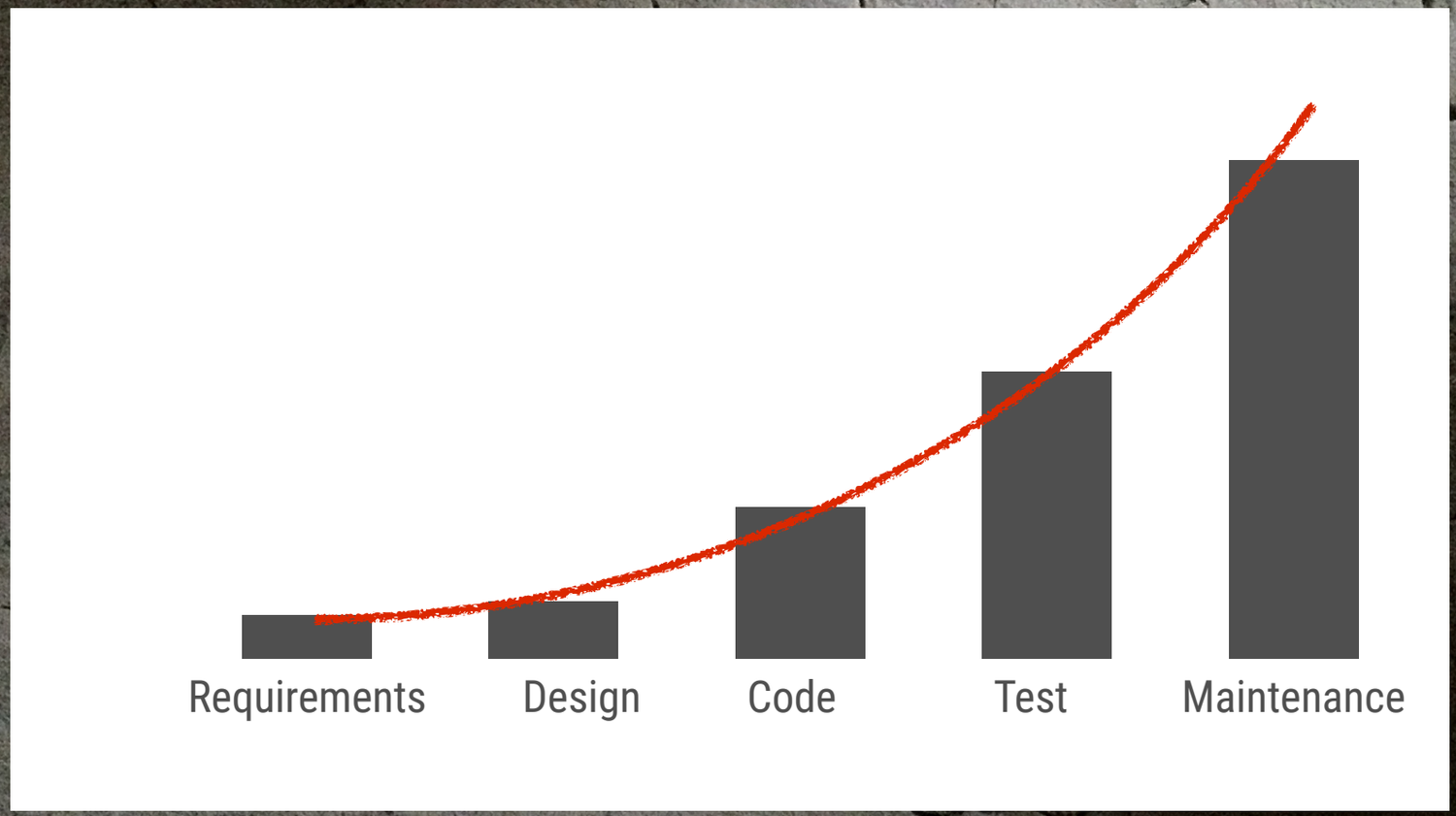
QUNIT - UNIT TESTS


Requirements
Design
Code
Test
Maintenance


modern topsoil
19th century wall
18th century cobbles
collapse of Medieval wall
Medieval wall
Medieval floor
Medieval garden soil
Saxon clay floor
Medieval pit
collapse of Roman building
Roman road
Roman wall
Roman road-ditch
Roman floor
Roman levelling dump
Iron Age soil
Iron Age ditch
post-hole with burnt post
natural subsoil


Requirements
Design
Code
Test
Maintenance

UNIT TESTING

- Das Unit Testing beschreibt Methoden zur Prüfung und Validierung, mit denen ein Programmierer testen kann, ob einzelne Units (Einheiten) des Quelltextes gebrauchsfertig (stabil) sind.

- Eine Unit ist der kleinste testbare Teil einer Anwendung.

WARUM SOLLTE CODE GETESTET WERDEN?

- In der Prä-Unittest Ära probiert man Code direkt in der Webseite aus.
- Man klickt ein wenig herum und sieht, ob irgendein Problem auftritt, das dann ge"fixt" werden kann.
- Diese Methode bringt einige Probleme mit sich.

ES IST WIRKLICH LÄSTIG!

- Klicken ist nicht wirklich einfach! Ist alles angeklickt? Jedes Mal? Nichts vergessen?
- Es ist sehr wahrscheinlich, dass beim Wiederholen eines Tests das eine oder andere ausgespart oder vergessen wird.

ALTE FEHLER BLEIBEN UNENTDECKT

- Ein Programmteil, das erfolgreich getestet wurde, enthält durchaus noch Fehler.
- Diese Fehler tauchen erst nach einem Refactoring auf.
- Sie werden Regressions = Rückschritt genannt.
- Solche Fehler sind nicht einfach zu finden.

UNITTEST BEHEBEN SOLCHE PROBLEME

- Unittests finden grundsätzliche Fehler in Codes. Auch dann, wenn man gar nicht danach sucht.
- Falls der Code nach einer Änderung nicht mehr korrekt arbeitet (auch wenn es vorher scheinbar so war), so wird einer Tests den Fehler anzeigen.

UNITTESTS TESTEN DIE ERGEBNISSE VON FUNKTIONEN

```
function add (a, b) {
   return (a+b);
}
var result = add (3,4);                 //    7 === 7
var result = add (3, "4");              // „34" !== 7
var result = add ("3", "4");            // „34" !== 7
var result = add ("drei", „vier");      // „dreivier“ !== 7
var result = add (drei, vier)           // drei is not defined
```

GETESTET WIRD DER RÜCKGABEWERT EINER FUNKTION

```
function getValueFromAttr(object, attribute) {
   return $(object).attr(attribute)
}
var url = getValueFromAttribute(event.target, ,href');

test('test_url_isPage.html', function() {
   strictEqual(getValueFromAttribute(event.target,'href'),
'page.html');
})
```

AUFBAU EINES UNITTESTS MIT QUNIT

QUNIT - HERKUNFT

- Qunit wurde ursprünglich von John Resig als Teil des jQuery Projektes entwickelt.
- Seit 2008 ist es ein eigenständiges Projekt innerhalb der jQuery Foundation und wird von Jörn Zäfferer geführt.
- Die Assertions von Qunit folgen der CommonJS Unit Testing Spezifikation. [http://wiki.commonjs.org/wiki/Unit_Testing/1.0]

AUFBAU EINES UNITTESTS

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width">
    <title>QUnit Example</title>
    <link rel="stylesheet"
          href="https://code.jquery.com/qunit/qunit-2.0.1.css">
</head>
<body>
    <div id="qunit"></div>
    <div id="qunit-fixture"></div>
    <script src="https://code.jquery.com/qunit/qunit-2.0.1.js"></script>
    <script src="my-test-collection.js"></script>
</body>
</html>
```

WIE WERDEN TESTS ANGELEGT?

AUFBAU UND TEST METHODEN

ASSERTIONS - BEHAUPTUNG

- Die Bausteine eines Unittest sind die sogenannten **assertions**, auf deutsch Behauptungen oder Annahmen

- Eine **assertion** ist eine Aussage, die das zurückgegebene Ergebnis einer codierten Funktion vorhersagt.

- Wird die Vorhersage nicht erfüllt, ist klar, dass in der Funktion etwas falsch ist.

OK()

- Der `ok()` Test
- Der einfachste Test mit einem boolschen Ergebnis.
- Der Test läuft erfolgreich durch, wenn das erste Argument **truthy** ist
- Er entspricht einem assert.ok() in CommonJS oder auch einem JUnit assertTrue().

EINE EINFACHE TEST - FUNKTION MIT OK()

```
// Funktion, die getestet werden soll
function add(a, b) {
    return a + b;
}
// Eine Testgruppe
QUnit.module("Module add() Tests", function () {

    // Eine Testreihe
    QUnit.test("test_add_is7.", function (assert) {
 // Die Tests
        assert.ok(add(3, 4) === 7, "3 + 4 == 7");
        assert.ok(add(1, 2) !== 7, "1 + 2 != 7");
    });

});
```

QUnit Example

Hide passed tests Check for Globals No try-catch

Module: All modules

Filter: Go

QUnit 2.0.1; Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.98 Safari/537.36

Tests completed in 7 milliseconds.
2 assertions of 2 passed, 0 failed.

1. **Module add() Tests: test_add_is7. (2)** Rerun 1 ms

1. 3 + 4 == 7 @ 0 ms
2. 1 + 2 != 7 @ 1 ms

Source: at Object. (http://localhost/jquery/qunit/my-test-collection.js:9:11)

WENN EIN TEST FEHLSCHLÄGT ...

```
// Funktion, die getestet werden soll
function add(a, b) {
    return a + b;
}
// Eine Testgruppe
QUnit.module("Module add() Tests", function () {

    // Eine Testreihe
    QUnit.test("test_add_is7.", function (assert) {
 // Die Tests
        assert.ok(add(3, 5) === 7, "3 + 4 == 7");
        assert.ok(!add(1, 2) !== 7, "1 + 2 != 7");
    });

});
```

QUnit Example

Hide passed tests Check for Globals No try-catch

Module: All modules

Filter: Go

QUnit 2.0.1; Mozilla/5.0 (Macintosh; Intel Mac OS X 10_12_1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/54.0.2840.98 Safari/537.36

Tests completed in 5 milliseconds.
1 assertions of 2 passed, 1 failed.

1. **Module add() Tests: test_add_is7. (1, 1, 2)** Rerun 1 ms

1. 3 + 4 == 7 @ 1 ms

Expected: `true`
Result: `false`
Source:
```
    at Object.<anonymous>
(http://localhost/jquery/qunit/my-test-
collection.js:11:16)
```

2. 1 + 2 != 7 @ 1 ms

Source: at Object. (http://localhost/jquery/qunit/my-test-collection.js:9:11)

VERGLEICHSTEST

```
// Eine Testgruppe
QUnit.module("module add() tests", function () {

    // Eine Testreihe
    QUnit.test("test_add_is7.", function (assert) {
 // Die Tests
        assert.equal(add(3, 4), 7, "3 + 4 == 7");

        assert.notEqual(add(3, 5), 7, "3 + 5 != 7");

        assert.strictEqual(add(3, 4), 7, "3 + 4 === 7");

        assert.notStrictEqual(add(3, 5), 7, "3 + 5 !== 7");
    });

});
```

OBJEKTE UND ARRAYS?

```
QUnit.module("module add() tests", function () {
    // Eine Testreihe
    QUnit.test("test_sequentialDataTypes", function (assert) {

        // Die Tests
        assert.equal( {}, {}, "schlägt fehl, zwei Objekte!");
        assert.equal( {a:1}, {a:1}, "schlägt fehl");
        assert.equal( [], [], "schlägt fehl, zwei Arrays!");
        assert.equal( [1], [1], "schlägt auch fehl");
    });
});
```

OBJEKTE UND ARRAYS MIT DEEPEQUAL()

```
QUnit.module("module add() object tests 2", function () {
    // Eine Testreihe
    QUnit.test("test_sequentialDataTypes", function (assert) {

        // Die Tests
        assert.deepEqual( {}, {}, „Objekte sind identisch.");
        assert.deepEqual( {a:1}, {a:1}, „Objekte sind identisch.");
        assert.deepEqual( [], [], "Arrays sind identisch.");
        assert.deepEqual( [1], [1], "Arrays sind identisch.");

        assert.notDeepEqual( [1], [2], "Arrays sind nicht identisch.");
    });
});
```

DEEPEQUAL()

- Eine tiefe rekursive Vergleichsbehauptung, die für sämtliche Datentypen herangezogen werden kann:
  - Einfache Typen
  - Arrays, Objekte
  - Auch für Reguläre Ausdrücke, Datumsobjekte und Funktionen.

PROPEQUAL(), NOTPROPEQUAL()

- Ein strikter Vergleich vom Objekteigenschaften.
- Dabei können auch Objekte mit verschiedenen Konstruktoren und Prototypen herangezogen werden.

PROPEQUAL()

```
// Ein Objekt Konstructor
function User( name, locations ) {
    this.name = name;
    this.locations = locations;
}

User.prototype.setName = function () {};
User.prototype.locations = [];

var user = new User("Michael", [„Cologne", "Stuttgart"]);
var object = {
      name : "Michael",
      locations : ["Cologne", "Stuttgart"]
};
```

PROPEQUAL()

```
QUnit.module("module add() object tests 3", function () {

  QUnit.test( "propEqual test", function( assert ) {
    assert.propEqual( user, object, "Strictly the same
properties without comparing objects constructors." );
  });

});
```

TRIPLE A

ANLEGEN EINES TESTS ODER EINER TESTREIHE.

GLIEDERN SIE TESTREIHEN IN MODULE

```
QUnit.module("module add() values", function () {});

QUnit.module("module add() value limits", function () {});

QUnit.module("module add() typing", function () {});
```

AUFBAU EINES TESTNAMENS

```
Test -> Modul -> Submodul->Funktionsname->Assertion

QUnit.test("test_add_is7", function (assert) {}
QUnit.test("test_add_equals7", function (assert) {}

QUnit.test("test_add_typeIsNumber", function (assert) {}
QUnit.test(„test_add_typeIsNotString", function (assert) {}


...
```

FÜR EIN ERGEBNIS SIND VIELE TESTS NÖTIG.

```
test_add_is7

// TEST 1: Integerwerte
   var a = 3;
   var b = 4;

// TEST 2 - Nachkommastellen
      var a = 4.333333333;
      var b = 2.666666667;

// TEST 3 - Große Zahlen
      var a =  212345678942345678189 7;
      var b = -212345678942345678189 0;
```

TRIPLE A

Ein Test sollte den drei A's folgen:
Assert   – Behauptung aufstellen
Arrange – Vorbereitungen treffen (set up)
Act   – Test ausführen

Nach der Testausführung muss aufgeräumt werden.
Teardown – Zurücksetzen der Testkonfiguration

AUFBAU EINES TESTS NACH TRIPLE A

```
QUnit.test(
   'test_app_myFunc_add_is7',
   function (assert) {
      // ASSERTION – Behauptung aufstellen
       var result = 7, a, b;

      // ARRANGE – Test vorbereiten (set up)
       a = 1; b = 2;

      // ACT – Test ausführen
       assert.equal( add(a,b), result, "3 + 4 === 7“);

    // TEARDOWN
    a = null; b= null;
  }
);
```

- Tests müssen einfach sein.
  Sie dürfen keine Fehler haben.

- Deshalb in Tests keine Kontrollstrukturen oder andere komplexe Programmablauf.

- Schreiben Sie einfach einen weiteren Test. Oder mehrere.

ASYNCHRONES TESTEN

- Bei AJAX Aufrufen oder Funktionen mit setTimeout() und setInterval() gibt es Zeitverzögerungen.
- Asynchrone Tests fangen dies ab.

ASYNC()

```
QUnit.test( "assert.async() test", function( assert ) {

  var done = assert.async(),
      input = $("#input").focus();

  setTimeout(function() {
    assert.equal( document.activeElement, input[0], "focus");
    done();
  });
});
```

ASYNC() MIT MEHREREN OPERATIONEN

```
QUnit.test( "two async calls", function( assert ) {
  assert.expect(2);

  var done1 = assert.async();
  var done2 = assert.async();

  setTimeout(function() {
    assert.ok( true, "test resumed from async operation 1" );
    done1();
  }, 500 );
  setTimeout(function() {
    assert.ok( true, "test resumed from async operation 2" );
    done2();
  }, 150);

});
```

ASYNC() MIT MEHREREN OPERATIONEN

```
QUnit.test( "multiple call done()", function( assert ) {
  assert.expect( 3 );
  var done = assert.async( 3 );

  setTimeout(function() {
    assert.ok( true, "first call done." );
    done();
  }, 500 );

  setTimeout(function() {
    assert.ok( true, "second call done." );
    done();
  }, 500 );

  setTimeout(function() {
    assert.ok( true, "third call done." );
    done();
  }, 500 );
});
```

MEHR FUNKTIONEN

QUnit.throws() - Test if a callback throws an exception, and optionally compare the thrown error.
QUnit.begin() - Register a callback to fire whenever the test suite begins.
QUnit.log() - Register a callback to fire whenever an assertion completes.
QUnit.moduleDone() - Register a callback to fire whenever a module ends.
QUnit.moduleStart() - Register a callback to fire whenever a module begins.
QUnit.testDone() - Register a callback to fire whenever a test ends.
QUnit.testStart() - Register a callback to fire whenever a test begins.
QUnit.config - Configuration for QUnit
QUnit.dump.parse() - Advanced and extensible data dumping for JavaScript
QUnit.extend() - Copy the properties defined by the mixin object into the target object
QUnit.push() - DEPRECATED: Report the result of a custom assertion
QUnit.stack() - Returns a single line string representing the stacktrace (call stack)
QUnit.only() - Adds a test to exclusively run, preventing all other tests from running.
QUnit.skip() - Adds a test like object to be skipped

http://api.qunitjs.com/