Unit Test Isolation with Dummies, Fakes, Stubs, Spies, and Mocks

What Are Test Doubles?

- Almost all code depends on and collaborates with other parts of the system.
- Those other parts of the system are not always easy to replicate in the unit test environment or would make tests slow if used directly.
- Test doubles are objects that are used in unit tests as replacements to the real production system collaborators.

Types of Test Doubles

- **Dummy** - Objects that can be passed around as necessary but do not have any type of test implementation and should never be used.
- **Fake** - These object generally have a simplified functional implementation of a particular interface that is adequate for testing but not for production.
- **Stub** - These objects provide implementations with canned answers that are suitable for the test.
- **Spies** - These objects provide implementations that record the values that were passed in so they can be used by the test.
- **Mocks** - These objects are pre-programmed to expect specific calls and parameters and can throw exceptions when necessary.

Mock Frameworks

- Most mock frameworks provide easy ways for automatically creating any of these types of test doubles ***at runtime***.
- They provide a fast means for creating mocking expectations for your tests.
- They can be much more efficient than implementing custom mock object of your own creation.
- Creating mock objects by hand can be tedious and error prone.

Sinon.JS

- Javascript Mocking Framework
- Works in NodeJS and a web browser
- Works well with Mocha and Chai

Creating a Spy

```
# Example
it('tests spies', function(){
  var callback = sinon.spy();
  prodFunction(callback);
  expect(callback).to.have.
    been.called();
});
```

- The most basic test double provided by Sinon is the spy.
- A spy is created by calling the sinon.spy method.
- A spy keeps track of:
  - How many times a function was called.
  - What parameters were passed to the function.
  - What value the function returned or if it threw.

Method Wrapping Spy

```
//Method Wrapping Spy
it('tests spies', function(){
  var tc = new TestClass();
  sinon.spy(tc, "testFunc");
  tc.testFunc();
  expect(tc.testFunc).to.have.
    been.called();
});
```

- Spies can be created in two fashions: either anonymous or wrapping a particular method.
- Anonymous spies are used to create fake functions that need to be spied on during testing.
- Method wrapping spies are created on existing functions such as class methods.

Spy API

- Sinon provides an extensive API for testing calls made to a spy. For example:
  - spy.callCount - The number of times the spy was called.
  - spy.called - True if the spy was called at least once.
  - spy.calledWith(arg1, arg2, …) - Spy was called with the specified arguments (and possibly others)
  - spy.returnValues - Array of return values made from the spied on function for each call to the function.
  - spy.threw - The spy threw an exception at least once.
  - Complete API available at: https://sinonjs.org/releases/v6.1.5/spies/

Sinon Stubs

```
//Sinon Stub
it('tests stub', function(){
  var tc = new TestClass();
  sinon.stub(tc, "testFunc");
  testCall(tc)
  expect(tc.testFunc).to.have.
    been.called();
});
```

- Sinon also provides an API for implementing stub test doubles.
- Stubs are like spies in that they can be anonymous or wrap existing functions.
- Stubs support the full Spy testing API.
- Stubs are different from spies in that they do NOT call the wrapped function
- Stubs allow you to modify the behavior of the stubbed function call.

Sinon Mocks

```
// Sinon Mocks
it('tests mock', function(){
  var tc = new TestClass();
  var mock = sinon.mock(tc);
  mock.expects('func').once();
  testCall(tc)
  mock.verify();
});
```

- Sinon also provides an API for creating mock objects.
- Sinon mocks provide all the capabilities of Sinon spies and stubs with the addition of pre-programmed expectations.
- A mock will verify that the specified expectations have occurred and if not will fail the test.

Sinon Mocks Expectations

- Sinon Mocks provide an extensive API of expectations that can be set. For example:
  - expectation.atLeast - The mock was called at least the specified number of times.
  - expectation.never - Verifies the mock was never called.
  - expectation.once - Verifies the mock was called once.
  - expectation.withArgs - The mock was called with the specified arguments and possibly others.
  - expectation.on(obj) - The mock was called with the specified object as “this”.
  - Complete API available at: https://sinonjs.org/releases/v6.1.5/mocks/

Sinon Cleanup

```
// Sinon Cleanup
afterEach(()=>{
  sinon.restore();
});
```

- Sinon creates all of its test doubles in a sandbox.
- Although you can create your own sandbox you will typically use Sinon's default sandbox.
- After each test the sandbox needs to be reset to clear out all the test doubles that were created by calling the sinon.restore method.

Sinon-Chai

```
//Method Wrapping Spy
it('tests spies', function(){
  var tc = new TestClass();
  sinon.spy(tc, "testFunc");
  tc.testFunc();
  tc.should.have.been.called();
});
```

- The Sinon-Chai library continues the BDD style expectations provide by Chai when using Sinon test doubles.
- Sinon-Chai provides an API that on your mocks that mimics the Chai expect and should APIs.
- This helps ensure your unit tests consistently follow the Chai BDD style of specifying expectations.