Mock Objects and the Mockito Testing Framework

Carl Veazey
CSCI 5828

Introduction

- Mock objects are a powerful testing pattern for verifying the behavior and interactions of systems.
- This presentation aims to introduce mock objects and related concepts such as Stubs and Testing Doubles.
- We'll also look in depth at the Mockito framework, a modern mock testing framework for Java.

Test Doubles

- Mock objects are a type of object that serves as a “test double”
- The use of test doubles is a testing pattern identified at xunitpatterns.com.
- Doubles allow testing the System Under Test (SUT) without having to test its Depended-on Component (DOC).
- The double provides a fake implementation of the DOC’s interface to the SUT.

Test Doubles

- Test doubles enable manipulation of the SUT through the fake DOC API, and allows us to make assertions about state or behavior in situations like:
  - No extant implementation of the DOC
  - The DOC's implementation is slow, e.g. a web service.
  - Critical inputs and outputs are not available through the SUT's public API

Test Doubles and Stubs

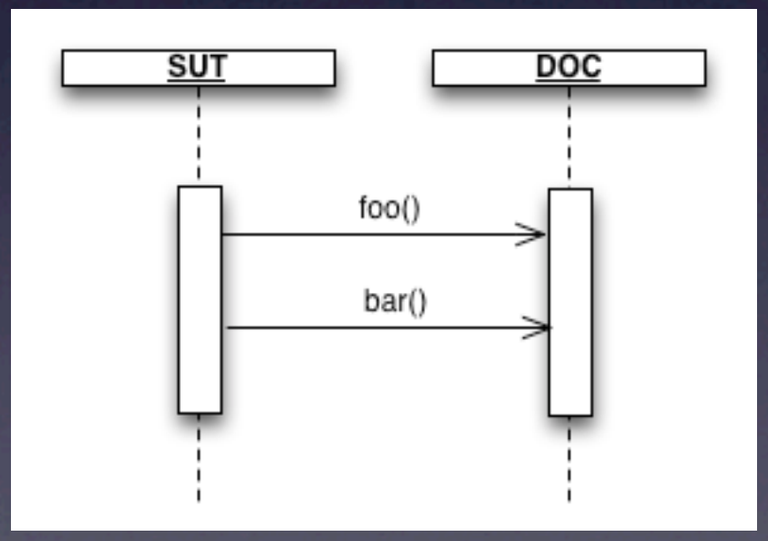
- Test stubs are types of test doubles that provide inputs to the SUT.
- Test fixtures can be loaded and returned by the stub through its implementation of the DOC's API.
- Allows assertions on SUT's state to be made based on contrived inputs from the DOC.

Mocks and Stubs

- A mock object often implements the same behavior as a stub, i.e. it can interact with the SUT through loading predetermined fixtures.
- However, its main purpose is to verify the SUT's interaction with the DOC. The programmer provides it with expectations about how the SUT should collaborate with it, and then verifies those expectations.

Mocks Verify Behavior

- Consider a system where the requirements for the interaction between SUT and DOC are described by the following sequence diagram:



State vs. Behavior

- Traditional unit tests test the state of the SUT after the test has been exercised.
- In contrast, tests with mock objects verify the way the SUT behaves with respect to its collaborators.
- State-based testing presents its assertions after the code has executed, while with mocks the assertions are presented as expectations before the SUT is exercised.

State vs. Behavior

- A more traditional test often takes the form of *setup - act - assert*. We assert a certain state of the system based on known inputs and actions.
- Mock-based testing often takes the form *expect - act - verify*. This allows one to focus solely on the behavior of and interactions between the SUT and its DOCs.

Mocks Verify Behavior

- A stub alone wouldn't be sufficient to test this interaction. The SUT has no change in state that can be observed after exercising the test.
- The mock then takes on the responsibility of verifying the requirement has been met.
- As mentioned before, the programmer provides the mock with a set of expectations and then verifies those expectations have been met.

Expectations

- The following pseudocode illustrates setting expectations and verifying them:

```
mySUT.setDOC(mockedDOC);
mockedDOC.expect.foo();
mockedDOC.expect.bar();
mySUT.doSomething();
mockedDOC.verify();
```

- The mocked DOC is told to expect both the methods foo() and bar() to be called on it. The test is exercised by calling doSomething() on the SUT, and then the mock's expectations are verified.

Applications for Mocks

- Anytime the SUT invokes side effects in any of its DOC's, this is a prime opportunity to use a mock, e.g. logging systems
- Verifying messages were sent in a particular order.
- Verifying certain messages were *not* sent.
- Verify certain arguments were passed without interfering with the SUT's public API.

Example Motivation

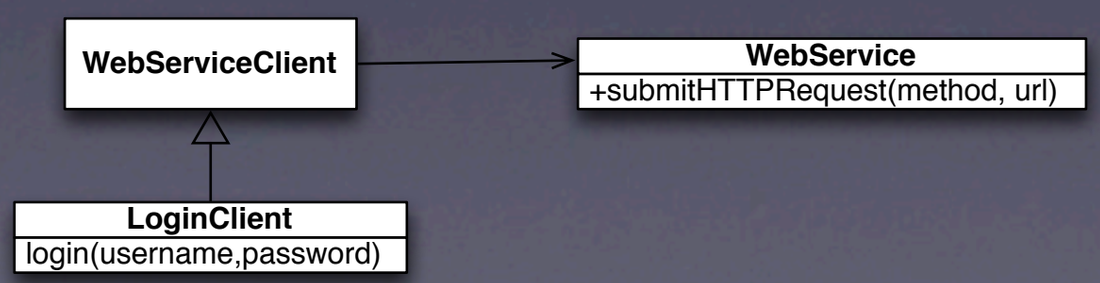
- Verifying interactions of your client side with a web service.
- You don't want to have you test code actually hit the web service, as your tests would be dependent on a slow external service. To get around this, we can create a mock of a web service.

Mock Web Service

- Suppose we have an application that makes requests to a web service to fulfill its functionality.
- We've created a WebService object in our application that forwards requests and responses to and from the network.
- Additionally, we have WebServiceClient objects that construct HTTP requests and submit them to the WebService.

Mock Web Service

- We'll want to write tests for the structure of the HTTP requests and the order in which they are made.
- We can create a mock of the WebService and verify that clients are asking it to handle the expected requests.
- Consider a WebServiceClient tasked with passing a user's credentials to the server:



Mock Web Service

- A test for such a case would look something like this:

```
WebServiceClient loginClient = createLoginClient();
WebService mockService = createMockWebService();
loginClient.setWebService(mockService);
mockService.expect().doHTTPRequest("GET",
                        "http://mywebservice.com/login?username=username&password=ae234fd298");
loginClient.login("username","ae234fd298");
mockService.verify();
```

- Notice that we haven't had to expose any state of the WebServiceClient, or expose any private methods constructing an HTTP request. We've purely tested the interaction between the client and the service.

Limitations and Criticisms

- Tests written with mock objects can become coupled tightly with the implementation of the system, making refactoring difficult.
- Assertions about the order and number of times methods are called and other details that may be brittle
- Not a substitute for integration tests.

Mock object frameworks

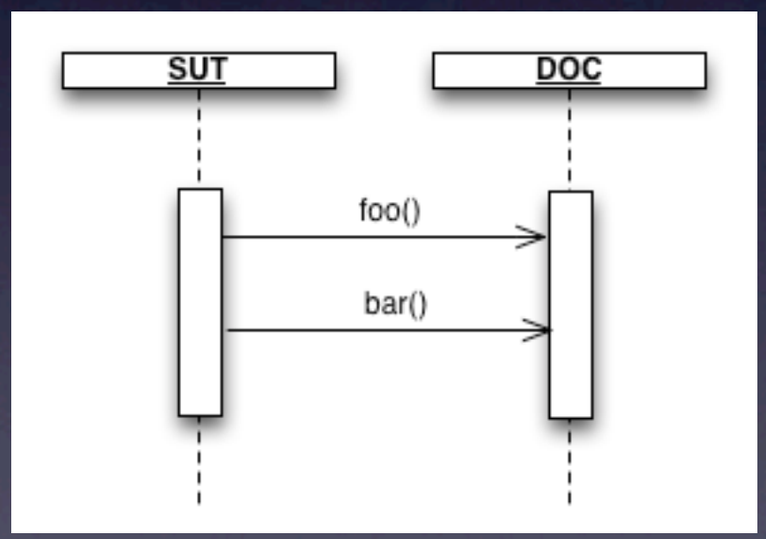
- Mock object frameworks exist for many OO languages and testing platforms.
- This presentation will take a look at Mockito, a modern mocking framework for Java.

Mockito

- Mockito is a lightweight, modern mocking framework. It is designed for cleanliness and speed in testing.
- Mockito takes a different approach from the test presented previously. Instead of the expect - act - verify test pattern, Mockito implicitly expects actions through the verification step, cutting down on the code you have to write.
- “There is only stubbing and verifications.”

Expectations and Verifications with Mockito

- Recall the contrived sequence diagram from before, and the pseudocode to test this interaction.




```
mySUT.setDOC(mockedDOC);
mockedDOC.expect.foo();
mockedDOC.expect.bar();
mySUT.doSomething();
mockedDOC.verify();
```

Expectations and Verifications with Mockito

- Here's how the previous code would look in Mockito:

```
Doc mockedDOC = mock(Doc.class);
mySUT.setDOC(mockedDOC);
mySUT.doSomething();
verify(mockedDoc.foo());
verify(mockedDoc.bar());
```

- A Mockito mock remembers all of its interactions so that they can be verified selectively and after the fact. As interactions become more complex, this reduces the amount of test code significantly.

Stubbing in Mockito (I)

- Stubs often important in the use of mocks to provide context to the SUT in order to provoke the correct interactions with the DOC.

- In Mockito, all method calls must be stubbed or they will return null or some other appropriate empty value.
- Stubbed methods can be used to throw exceptions, forward messages, or return values.

Stubbing In Mockito (II)

- Let's look at an example of stubbing a Java List. Obviously this is a contrived example but it illustrates the ability to inject arbitrary return values or force other code to be executed.

```
List mockedList = mock(List.class);

when(mockedList.get(0)).thenReturn("1st element");
when(mockedList.get(1)).thenThrow(new RuntimeException());

//this prints "1st element"
System.out.println(mockedList.get(0));

//this throws an execption
System.out.println(mockedList.get(1));

//this prints null as we didn't stub for the 2 argument
System.out.println(mockedList.get(2));
```

Stubbing in Mockito (III)

- Stub methods can also invoke blocks of code instead of returning prefabricated answers, using the thenAnswer() method.

- An example from Mockito documentation:

```
1 when(mock.someMethod(10)).thenAnswer(new Answer<Integer>() {
2     public Integer answer(InvocationOnMock invocation) throws Throwable {
3         return (Integer) invocation.getArguments()[0];
4     }
5 }
```

Verifying Arguments

- Stubbing is of course secondary to any mock object implementation. We are interested in verifying the interaction between the SUT and the DOC, so Mockito gives us tools to verify that certain methods with certain arguments were called by the DOC.
- Mockito can verify that specific or generic arguments were passed to the mock.

Verifying Arguments (II)

- Verifying a specific argument:

```
Doc mockedDOC = mock(Doc.class);
mySUT.setDOC(mockedDOC);
mySUT.doSomething(); //expected to call foo("specific argument")
verify(mockedDoc.foo("specific argument"));
```

- More generic arguments can also be expected using the Matchers class, which provides a rich variety of verification functions.

```
Doc mockedDOC = mock(Doc.class);
mySUT.setDOC(mockedDOC);
mySUT.doSomething(); //expected to call foo() with some string argument
verify(mockedDoc.foo(anyString()));
```

Verifying Arguments(III)

- We're not limited to built in Java types; we can create our own custom argument matchers.

- From the Mockito documentation:

```
class IsListOfTwoElements extends ArgumentMatcher<List> {
   public boolean matches(Object list) {
       return ((List) list).size() == 2;
   }
}

List mock = mock(List.class);

when(mock.addAll(argThat(new IsListOfTwoElements()))).thenReturn(true);

mock.addAll(Arrays.asList("one", "two"));

verify(mock).addAll(argThat(new IsListOfTwoElements()));
```

Verifying Behavior

- Beyond just matching arguments, Mockito can verify the number of times methods were called on the mocked object and the order methods were called.
- It can also verify that methods were never called.
- The verify() method can take an optional argument of an instance of a VerificationMode implementation.

Verification Modes

- Pass the return value of the times() method to verify() in order to verify the number of times a method was called:

```
List mockedList = mock(List.class);

mockedList.add("add three times");
mockedList.add("add three times");
mockedList.add("add three times");

// this will pass
verify(mockedList, times(3).add("add three times"));

mockedList.add("add twice");

// this will fail
verify(mockedList, times(2)).add("add twice");
```

Verification Modes

- We don't have to know a specific number, we can use atLeastOnce(), atLeast(), and atMost() as well.

- From the docs:

```
//verification using atLeast()/atMost()
verify(mockedList, atLeastOnce()).add("three times");
verify(mockedList, atLeast(2)).add("five times");
verify(mockedList, atMost(5)).add("three times");
```

Order Verification

- Similar to verification modes, the InOrder class allows us to make assertions about the order methods were called.

- In this case, verify() is called on the InOrder object itself, with invocations of verify() called in the expected order.

```
List mockedList = mock(List.class);

mockedList.add("first invocation");
mockedList.add("second invocation");

InOrder inorder = inOrder(mockedList);
inorder.verify(mockedList).add("first invocation");
inorder.verify(mockedList).add("second invocation");
```

More Verification

- We can make sure a method was never called on a mock using the VerificationMode returned by never().
- We can verify that there was no interaction with a mock using the verifyZeroInteractions() method, passing the mock we are asserting as the argument.
- To verify there were no unexpected interactions with the DOC, we can call the verifyNoMoreInteractions() method.

More Mockito

- Mockito offers more features than would fit in this presentation, including advanced stubbing APIs.
- Find out more with the Mockito documentation at http://docs.mockito.googlecode.com/hg/latest/org/mockito/Mockito.html or at the project homepage: http://code.google.com/p/mockito/

Further Reading

- xunitpatterns.com provides information about Mock Objects and other related Testing Doubles.
- Martin Fowler explains the difference between Mocks and Stubs in the article "Mocks Aren't Stubs". http://martinfowler.com/articles/mocksArentStubs.html
- An older but still informative article from IBM on testing with mocks in Java: http://www.ibm.com/developerworks/java/library/j-mocktest/index.html

Conclusion

- We looked at Testing Doubles, Stubs, and Mock Objects.
- Saw how testing with mocks verifies behaviors and interactions, while traditional testing verifies state.
- Dug into the Mockito framework.
- Any questions? Email me at carl.veazey@gmail.com! Thanks!