**Mocks And Stubs - Understanding Test Doubles With Mockito**

**Introduction**

A common thing I come across is that teams using a mocking framework assume they are mocking.

They are not aware that Mocks are just one of a number of 'Test Doubles' which Gerard Meszaros has categorised at xunitpatterns.com.

It’s important to realise that each type of test double has a different role to play in testing. In the same way that you need to learn different patterns or refactoring’s, you need to understand the primitive roles of each type of test double. These can then be combined to achieve your testing needs.

I'll cover a very brief history of how this classification came about, and how each of the types differs.

I'll do this using some short, simple examples in Mockito.

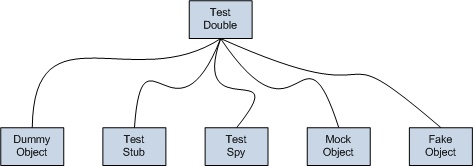
**A Very Brief History**

For years people have been writing lightweight versions of system components to help with testing. In general it was called stubbing. In 2000' the article 'Endo-Testing: Unit Testing with Mock Objects' introduced the concept of a Mock Object. Since then Stubs, Mocks and a number of other types of test objects have been classified by Meszaros as Test Doubles.

This terminology has been referenced by Martin Fowler in "Mocks Aren't Stubs" and is being adopted within the Microsoft community as shown in "Exploring The Continuum of Test Doubles"

A link to each of these important papers are shown in the reference section.

**Categories of test doubles**



The diagram above shows the commonly used types of test double. The following URL gives a good cross reference to each of the patterns and their features as well as alternative terminology.

<http://xunitpatterns.com/Test%20Double.html>

**Mockito**

Mockito is a test spy framework and it is very simple to learn. Notable with Mockito is that expectations of any mock objects are not defined before the test as they sometimes are in other mocking frameworks. This leads to a more natural style(IMHO) when beginning mocking.

The following examples are here purely to give a simple demonstration of using Mockito to implement the different types of test doubles.

There are a much larger number of specific examples of how to use Mockito on the website.

<http://docs.mockito.googlecode.com/hg/latest/org/mockito/Mockito.html>

**Test Doubles with Mockito**

Below are some basic examples using Mockito to show the role of each test double as defined by Meszaros.

I’ve included a link to the main definition for each so you can get more examples and a complete definition.

**Dummy Object**

<http://xunitpatterns.com/Dummy%20Object.html>

This is the simplest of all of the test doubles. This is an object that has no implementation which is used purely to populate arguments of method calls which are irrelevant to your test.

For example, the code below uses a lot of code to create the customer which is not important to the test.

The test couldn't care less which customer is added, as long as the customer count comes back as one.

**public** **Customer** createDummyCustomer() {

**County** county = **new** **County**("Essex");

**City** city = **new** **City**("Romford", county);

**Address** address = **new** **Address**("1234 Bank Street", city);

**Customer** customer = **new** **Customer**("john", "dobie", address);

**return** customer;

}

@Test

**public** **void** addCustomerTest() {

**Customer** dummy = createDummyCustomer();

**AddressBook** addressBook = **new** **AddressBook**();

addressBook.addCustomer(dummy);

assertEquals(1, addressBook.getNumberOfCustomers());

}

We actually don't care about the contents of customer object - but it is required. We can try a null value, but if the code is correct you would expect some kind of exception to be thrown.

@Test(expected=**Exception**.**class**)

**public** **void** addNullCustomerTest() {

**Customer** dummy = **null**;

**AddressBook** addressBook = **new** **AddressBook**();

addressBook.addCustomer(dummy);

}

To avoid this we can use a simple Mockito dummy to get the desired behaviour.

@Test

**public** **void** addCustomerWithDummyTest() {

**Customer** dummy = mock(**Customer**.**class**);

**AddressBook** addressBook = **new** **AddressBook**();

addressBook.addCustomer(dummy);

**Assert**.assertEquals(1, addressBook.getNumberOfCustomers());

}

It is this simple code which creates a dummy object to be passed into the call.

**Customer** dummy = mock(**Customer**.**class**);

Don't be fooled by the mock syntax - the role being played here is that of a dummy, not a mock.

It's the role of the test double that sets it apart, not the syntax used to create one.

This class works as a simple substitute for the customer class and makes the test very easy to read.

**Test stub**

<http://xunitpatterns.com/Test%20Stub.html>

The role of the test stub is to return controlled values to the object being tested. These are described as indirect inputs to the test.  Hopefully an example will clarify what this means.

Take the following code

**public** **class** **SimplePricingService** **implements** **PricingService**

{

**PricingRepository** repository;

**public** **SimplePricingService**(**PricingRepository** pricingRepository) {

**this**.repository = pricingRepository;

}

@Override

**public** **Price** priceTrade(**Trade** trade) {

**return** repository.getPriceForTrade(trade);

}

@Override

**public** **Price** getTotalPriceForTrades(**Collection**<trade> trades) {

**Price** totalPrice = **new** **Price**();

**for** (**Trade** trade : trades)

{

**Price** tradePrice = repository.getPriceForTrade(trade);

totalPrice = totalPrice.add(tradePrice);

}

**return** totalPrice;

}

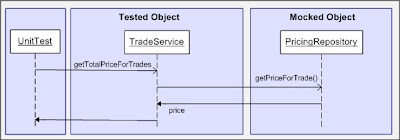
</trade>

The SimplePricingService has one collaborating object which is the trade repository. The trade repository provides trade prices to the pricing service through the getPriceForTrade method.

For us to test the businees logic in the SimplePricingService, we need to control these indirect inputs

i.e. inputs we never passed into the test.

This is shown below.



In the following example we stub the PricingRepository to return known values which can be used to test the business logic of the SimpleTradeService.

@Test

**public** **void** testGetHighestPricedTrade() **throws** **Exception** {

**Price** price1 = **new** **Price**(10);

**Price** price2 = **new** **Price**(15);

**Price** price3 = **new** **Price**(25);

**PricingRepository** pricingRepository = mock(**PricingRepository**.**class**);

**when**(pricingRepository.getPriceForTrade(any(**Trade**.**class**)))

.thenReturn(price1, price2, price3);

**PricingService** service = **new** **SimplePricingService**(pricingRepository);

**Price** highestPrice = service.getHighestPricedTrade(getTrades());

assertEquals(price3.getAmount(), highestPrice.getAmount());

}

**Saboteur Example**

There are 2 common variants of Test Stubs: Responder’s and Saboteur's.

Responder's are used to test the happy path as in the previous example.

A saboteur is used to test exceptional behaviour as below.

@Test(expected=**TradeNotFoundException**.**class**)

**public** **void** testInvalidTrade() **throws** **Exception** {

**Trade** trade = **new** **FixtureHelper**().getTrade();

**TradeRepository** tradeRepository = mock(**TradeRepository**.**class**);

**when**(tradeRepository.getTradeById(anyLong()))

.thenThrow(**new** **TradeNotFoundException**());

**TradingService** tradingService = **new** **SimpleTradingService**(tradeRepository);

tradingService.getTradeById(trade.getId());

}

**Mock Object**

<http://xunitpatterns.com/Mock%20Object.html>

Mock objects are used to verify object behaviour during a test. By object behaviour I mean we check that the correct methods and paths are excercised on the object when the test is run.

This is very different to the supporting role of a stub which is used to provide results to whatever you are testing.

In a stub we use the pattern of defining a return value for a method.

**when**(customer.getSurname()).thenReturn(surname);

In a mock we check the behaviour of the object using the following form.

verify(listMock).add(s);

Here is a simple example where we want to test that a new trade is audited correctly.

Here is the main code.

**public** **class** **SimpleTradingService** **implements** **TradingService**{

**TradeRepository** tradeRepository;

**AuditService** auditService;

**public** **SimpleTradingService**(**TradeRepository** tradeRepository,

**AuditService** auditService)

{

**this**.tradeRepository = tradeRepository;

**this**.auditService = auditService;

}

**public** **Long** createTrade(**Trade** trade) **throws** **CreateTradeException** {

**Long** id = tradeRepository.createTrade(trade);

auditService.logNewTrade(trade);

**return** id;

}

The test below creates a stub for the trade repository and mock for the AuditService

We then call verify on the mocked AuditService to make sure that the TradeService calls it's

logNewTrade method correctly

@Mock

**TradeRepository** tradeRepository;

@Mock

**AuditService** auditService;

@Test

**public** **void** testAuditLogEntryMadeForNewTrade() **throws** **Exception** {

**Trade** trade = **new** **Trade**("Ref 1", "Description 1");

**when**(tradeRepository.createTrade(trade)).thenReturn(anyLong());

**TradingService** tradingService

= **new** **SimpleTradingService**(tradeRepository, auditService);

tradingService.createTrade(trade);

verify(auditService).logNewTrade(trade);

}

The following line does the checking on the mocked AuditService.

verify(auditService).logNewTrade(trade);

This test allows us to show that the audit service behaves correctly when creating a trade.

**Test Spy**

<http://xunitpatterns.com/Test%20Spy.html>

It's worth having a look at the above link for the strict definition of a Test Spy.

However in Mockito I like to use it to allow you to wrap a real object and then verify or modify it's behaviour to support your testing.

Here is an example were we check the standard behaviour of a List. Note that we can both verify that the add method is called and also assert that the item was added to the list.

@Spy

**List**<string> listSpy = **new** **ArrayList**<string>();

@Test

**public** **void** testSpyReturnsRealValues() **throws** **Exception** {

**String** s = "dobie";

listSpy.add(**new** **String**(s));

verify(listSpy).add(s);

assertEquals(1, listSpy.size());

}

</string></**string**>

Compare this with using a mock object where only the method call can be validated. Because we only mock the behaviour of the list, it does not record that the item has been added and returns the default value of zero when we call the size() method.

@Mock

**List**<string> listMock = **new** **ArrayList**<string>();

@Test

**public** **void** testMockReturnsZero() **throws** **Exception** {

**String** s = "dobie";

listMock.add(**new** **String**(s));

verify(listMock).add(s);

assertEquals(0, listMock.size());

}

</string></**string**>

Another useful feature of the testSpy is the ability to stub return calls. When this is done the object will behave as normal until the stubbed method is called.

In this example we stub the get method to always throw a RuntimeException. The rest of the behaviour remains the same.

@Test(expected=**RuntimeException**.**class**)

**public** **void** testSpyReturnsStubbedValues() **throws** **Exception** {

listSpy.add(**new** **String**("dobie"));

assertEquals(1, listSpy.size());

**when**(listSpy.**get**(anyInt())).thenThrow(**new** **RuntimeException**());

listSpy.**get**(0);

}

In this example we again keep the core behaviour but change the size() method to return 1 initially and 5 for all subsequent calls.

**public** **void** testSpyReturnsStubbedValues2() **throws** **Exception** {

**int** size = 5;

**when**(listSpy.size()).thenReturn(1, size);

**int** mockedListSize = listSpy.size();

assertEquals(1, mockedListSize);

mockedListSize = listSpy.size();

assertEquals(5, mockedListSize);

mockedListSize = listSpy.size();

assertEquals(5, mockedListSize);

}

This is pretty Magic!

**Fake Object**

<http://xunitpatterns.com/Fake%20Object.html>

Fake objects are usually hand crafted or light weight objects only used for testing and not suitable for production. A good example would be an in-memory database or fake service layer.

They tend to provide much more functionality than standard test doubles and as such are probably not usually candidates for implementation using Mockito. That’s not to say that they couldn’t be constructed as such, just that its probably not worth implementing this way.