Mockito in six easy examples

Published by gojko at 1:41 pm under articles

Mockito is a fantastic mock library for Java. I'm fascinated by how easy it is to use, compared to other things out there both in the Java and .NET world. Here is everything you need to know to get started in six really easy examples.

First of all, get mockito from http://mockito.org/. Almost everything really interesting can be imported with the org.mockito.Mockito class (or a static import of its methods, which I'll use in this post). So let's get right into it.

To create a stub (or a mock), use mock(class). Then use when(mock).thenReturn(value) to specify the stub value for a method. If you specify more than one value, they will be returned in sequence until the last one is used, after which point the last specified value gets returned. (So to have a method return the same value always, just specify it once). For example:

```
import static org.mockito.Mockito.*;
import static org.junit.Assert.*;
import java.util.Iterator;
import org.junit.Test;
....
        @Test
        public void iterator_will_return_hello_world(){
                //arrange
                Iterator i=mock(Iterator.class);
                when(i.next()).thenReturn("Hello").thenReturn("World");
                //act
                String result=i.next()+" "+i.next();
                //assert
                assertEquals("Hello World", result);
        }
```

This example creates a mock iterator and makes it return "Hello" the first time method next() is called. Calls after that return "World". Then we can run normal assertions.

Stubs can also return different values depending on arguments passed into the method. For example:

```
        @Test
        public void with_arguments(){
                Comparable c=mock(Comparable.class);
                when(c.compareTo("Test")).thenReturn(1);
                assertEquals(1,c.compareTo("Test"));
        }
```

This creates a stub Comparable object and returns 1 if it is compared to a particular String value ("Test" in this case). If the method has arguments but you really don't care what gets passed or cannot predict it, use anyInt() (and alternative values for other types). For example:

```
        @Test
        public void with_unspecified_arguments(){
                Comparable c=mock(Comparable.class);
                when(c.compareTo(anyInt())).thenReturn(-1);
                assertEquals(-1,c.compareTo(5));
        }
```

This stub comparable returns -1 regardless of the actual method argument. With void methods, this gets a bit tricky as you can't use them in the when() call. The alternative syntax is doReturn(result).when(mock_object).void_method_call(); Instead of returning, you can also use .thenThrow() or doThrow() for void methods. For example:

```
        @Test(expected=IOException.class)
        public void OutputStreamWriter_rethrows_an_exception_from_OutputStream()
                throws IOException{
                OutputStream mock=mock(OutputStream.class);
                OutputStreamWriter osw=new OutputStreamWriter(mock);
                doThrow(new IOException()).when(mock).close();
                osw.close();
        }
```

This example throws an IOException when the mock OutputStream close method is called. We verify easily that the OutputStreamWriter rethrows the exception of the wrapped output stream. To verify actual calls to underlying objects (typical mock object usage), we can use verify(mock_object).method_call; For example:

```
        @Test
        public void OutputStreamWriter_Closes_OutputStream_on_Close()
                 throws IOException{
                OutputStream mock=mock(OutputStream.class);
                OutputStreamWriter osw=new OutputStreamWriter(mock);
                osw.close();
                verify(mock).close();
```

```
;
```

This example will verify that OutputStreamWriter propagates the close method call to the wrapped output stream. You can use arguments on methods and matchers such as anyInt() similar to the previous example. Note that you can't mix literals and matchers, so if you have multiple arguments they all have to be either literals or matchers. use eq(value) matcher to convert a literal into a matcher that compares on value. Mockito comes with lots of matchers already built in, but sometimes you need a bit more flexibility. For example, OutputStreamWriter will buffer output and then send it to the wrapped object when flushed, but we don't know how big the buffer is upfront. So we can't use equality matching. However, we can supply our own matcher:

```
@Test
public void OutputStreamWriter_Buffers_And_Forwards_To_OutputStream()
        throws IOException{
        OutputStream mock=mock(OutputStream.class);
        OutputStreamWriter osw=new OutputStreamWriter(mock);
        osw.write('a');
        osw.flush();
        // can't do this as we don't know how long the array is going to be
        // verify(mock).write(new byte[]{'a'},0,1);

        BaseMatcher arrayStartingWithA=new BaseMatcher(){
                @Override
                public void describeTo(Description description) {
                        // nothing
                }
                // check that first character is A
                @Override
                public boolean matches(Object item) {
                        byte[] actual=(byte[]) item;
                        return actual[0]=='a';
                }
        };
        // check that first character of the array is A, and that the other two arguments are 0 and 1
        verify(mock).write(argThat(arrayStartingWithA), eq(0),eq(1));
}
```

That's it – all you need to know to get started. Now go forth and refactor all that easymock ugliness from your projects.