Introduction to Software Engineering

Integration Testing with Mockito

Rami Sayar @ramisayar

GitHub Talks If you find a mistake, don't hesitate to fork the repository, clone it to your local machine, edit the mistake, `git commit -a -m` and then press the pull request button to notify me.

Introduction

The purpose of these notes is to introduce you to integration testing with Mockito. Mocking is a technique for providing a "mock" representation of resources such as dependencies or "stubs". Mock objects will act as stubs and allow us to verify the interactions between different classes. The Mockito library allows us to create mock objects easily without tediously writing out classes with the appropriate interface. It is one of the easier mocking frameworks to use.

Mocking a Class

Taking a look back at our Bixi case study and design patterns tutorial, at one point we wanted to create a chain of responsibility to relay messages from a manager to an employee. It's difficult to write a unit-test to verify the behaviour of a chain of responsibility pattern without an integration test.

Let's recall our example from the design patterns tutorial.

```
abstract class Supervisor {
    protected Supervisor next;
    public void setNext(Supervisor next) {
        this.next = next;
    }
    public void inform(String msg) {
        if (next != null) {
            next.inform(msg);
        }
    }
}

class Manager extends Supervisor {
    public void inform(String msg) {
        // Do something!
        super.inform(msg)
    }
}

class Executive extends Supervisor {
    public void inform(String msg) {
        // Do something!
        super.inform(msg)
    }
}

class Mayor extends Supervisor {
    public void inform(String msg) {
        // Do something!
        super.inform(msg)
    }
}
// Creating a Chain
Manager manager = new Manager();
Executive exec = new Executive();
Mayor mayor = new Mayor();
manager.setNext(exec);
exec.setNext(mayor);
```

In this case, we want to ensure that the message received by the mayor is the current one. Off course, in our test we can inherit from supervisor and verify the rest of each method manually. However, as you can imagine, if the `Supervisor` class was more complicated, we would have to tediously implement many methods manually. Instead, we want to mock a `TestSupervisor` easily. We can do this with mockito.

```
import static org.mockito.Mockito.*;

...

@Test
public void testInteraction(){
    // Creating a mock object.
    Supervisor superv = mock(Supervisor.class);

    Manager manager = new Manager();
    Executive exec = new Executive();
    Mayor mayor = new Mayor();
    manager.setNext(exec);
    exec.setNext(mayor);
    mayor.setNext(superv);

    // Defining behaviour
    when(superv.inform(anyString())).thenReturn(true);

    String m = "message";
    manager.inform(m);

    // Verifying behaviour
    verify(superv, times(1)).inform(m);
}
```

As you can see from the above code, we have created a mock object, defined its behaviour and verified its behaviour.

Creating a mock object is fairly straightforward, you can use the mock method or `@Mock`. However, if you use the annotation method, you need to execute `MockitoAnnotations.initMocks(testClass)` in a test runner or before the tests run.

Defining behaviour to be able to verify the interactions follows the 'where x runs return y' pattern. You can specify the arguments for x using any of the below *argument matches*:

- any() - matches any object or null
- anyInt(), anyString(), etc. - matches primitives or their wrappers
- eq(int value), eq(String value), etc. - argument matches the given value
- notNull() - not a null argument
- same(T value) - object argument is the same as the given value
- anyCollection(), anyCollectionOf(java.lang.Class) - matches any collection (generic friendly)
- matches(String regexp) - matches a given regular expression

Verifing the behaviour is as simple as calling verify on the object mock how many times a method was called with a specific parameter. In this case, we want to verify that inform was called once with the parameter m. You can verify that a method is `never()` called, `verifyNoMoreInteractions` or `verifyZeroInteractions(mock1,mock2)`. You can also pass anyString() as the parameter to the method called if you don't care what parameter was passed to it.

Conclusion

Using mock objects to create stubs and conduct integration tests is extremely powerful. You can pretty much develop all the unit tests and integration tests using mock objects without ever writing the actual code. Its power lies in allowing you to write tests first and conduct proper test-driven development. Mockito can do a lot more and I highly recommend you take a look at the Mockito API.

Reference

Mockito API Mockito - Integration testing made easier