45426: Teste e Qualidade de Software

Unit tests and mock objects

Ilídio Oliveira

v2020-02-18

universidade de aveiro
departamento de eletrónica,
telecomunicações e informática
deti

Learning objectives

Explain the practice of “tests in isolation”

Distinguish between mocks and stubs

For a given test case, identify which services should be mocked

Read and write (simple) unit tests using JUnit and Mockito

Unit testing assumes using units in isolation

Unit tests verify the "local" contract

StockPortfolio add/remove/find…

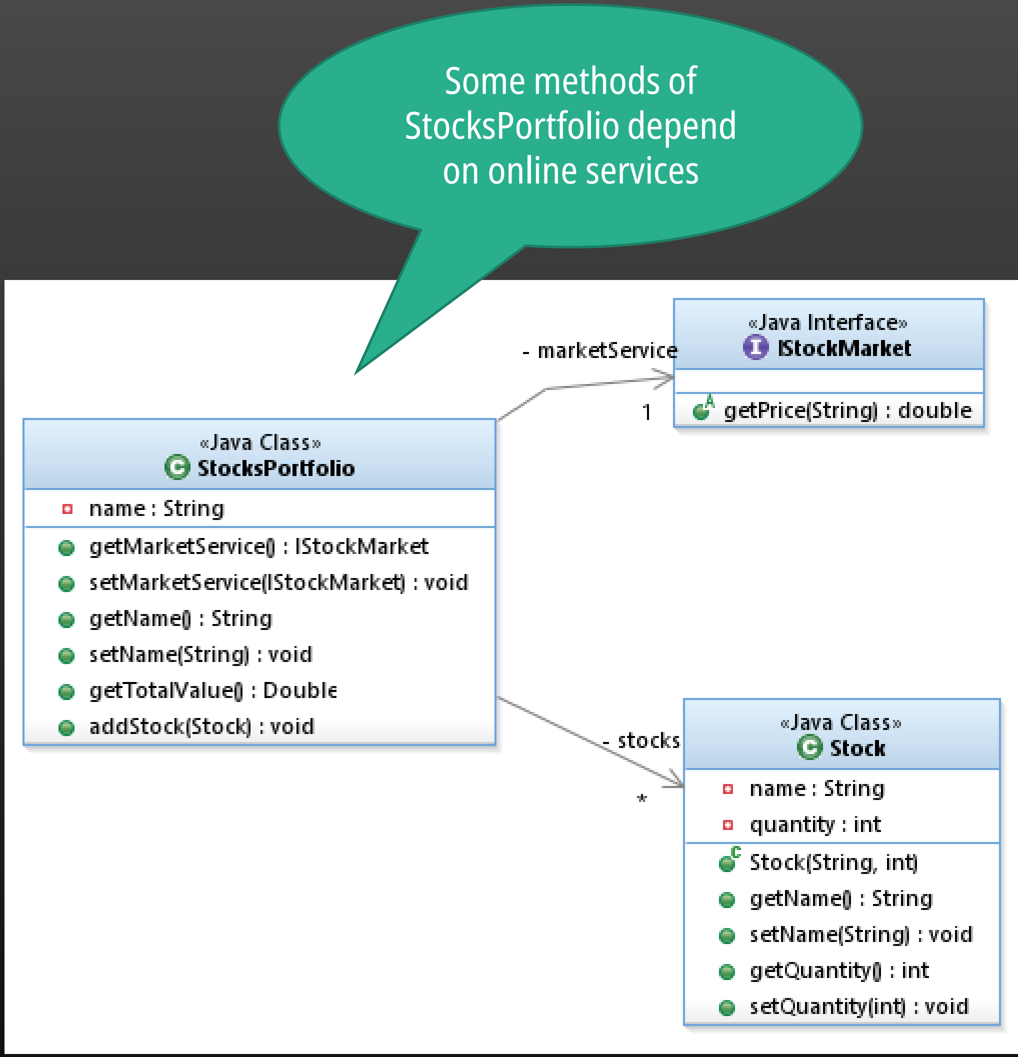
How about testing objects that require collaboration from others?

StockPortfolio queries StockMarket to update rates

To keep the isolation:

"fake" the behavior of the remote/real object/service

Just enough logic to get the tests done



Faking remote object behavior strategies

Stubs

"Mini-implementation", provides canned answers to calls made during the test.

Usually not responding at all to anything outside what's programmed in for the test

Mocks

Objects pre-programmed with expectations, i.e. specification of the interactions they are expected to receive.

Verification will compare calls received against expectations.

In-container testing

containers are activated (by the test runner) to enable the testing environment

Requires mechanisms to deploy and execute tests in a container

Stubs approach: drawbacks

Stubs require implementing the same logic as the systems they are replacing

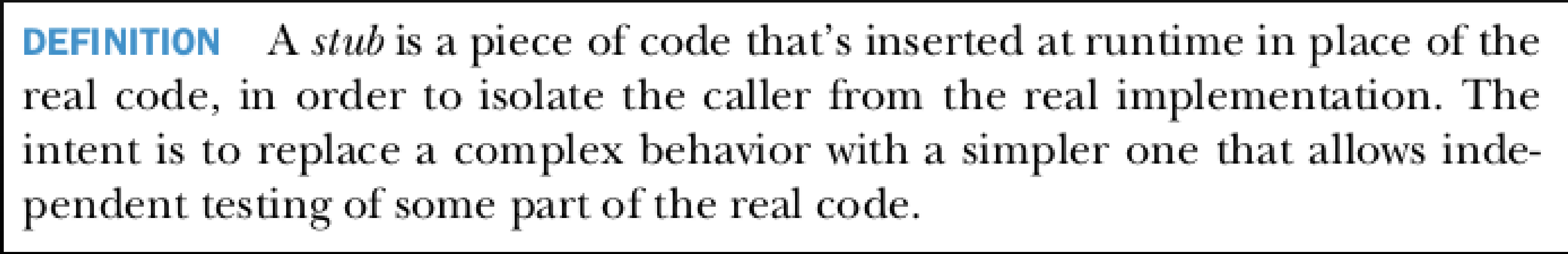
Often complex… stubs themselves need debugging!

Not very refactoring-friendly

Stubs don’t lend themselves well to fine-grained unit testing.

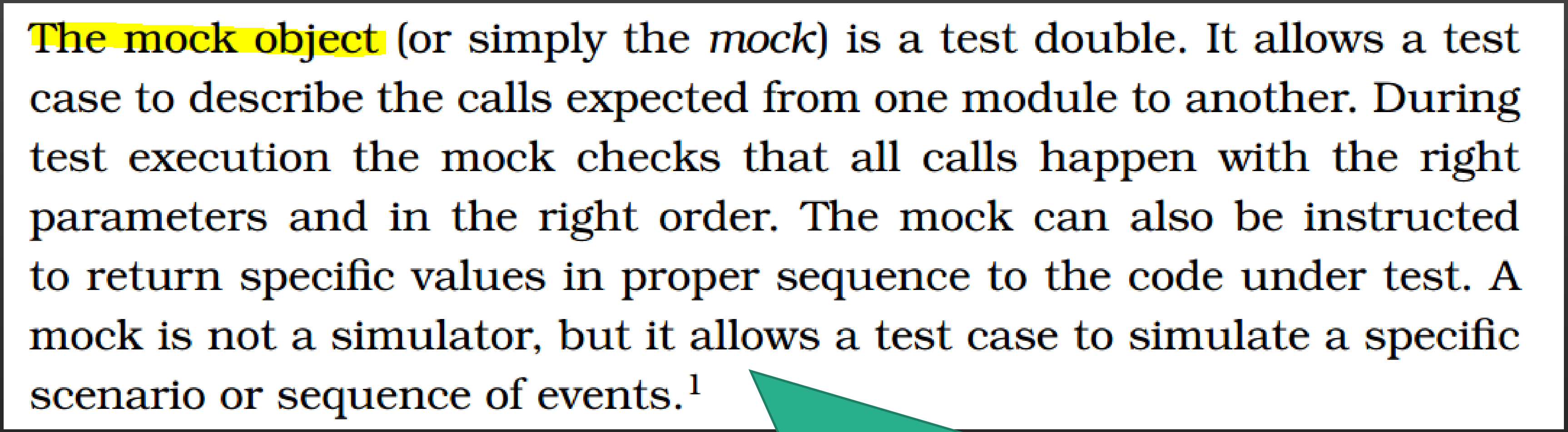
Use stubs to replace a full-blown external system

a file system, a connection to a server, a database, …



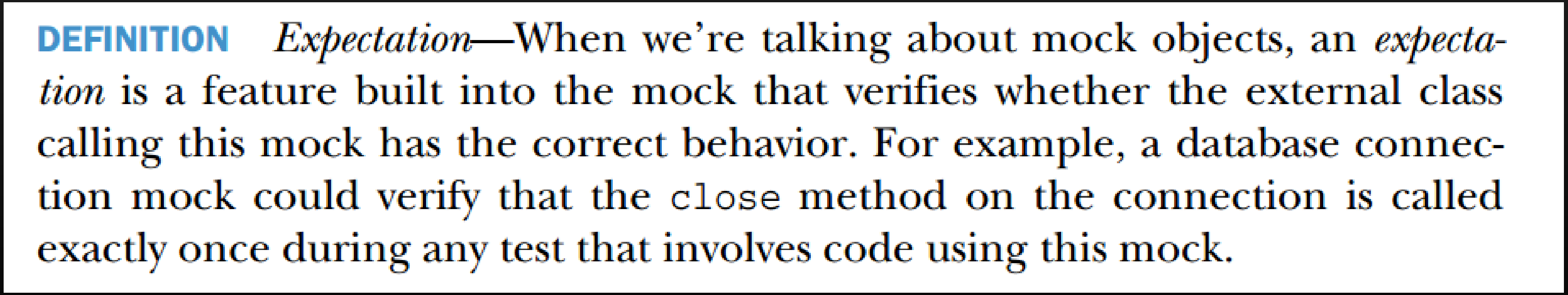
DEFINITION A *stub* is a piece of code that’s inserted at runtime in place of the real code, in order to isolate the caller from the real implementation. The intent is to replace a complex behavior with a simpler one that allows independent testing of some part of the real code.

Mock objects approach



The mock object (or simply the *mock*) is a test double. It allows a test case to describe the calls expected from one module to another. During test execution the mock checks that all calls happen with the right parameters and in the right order. The mock can also be instructed to return specific values in proper sequence to the code under test. A mock is not a simulator, but it allows a test case to simulate a specific scenario or sequence of events.[1]

The mock is a dumb object that does what the test tells it to do (behavior defined by expectations in the test case).



DEFINITION *Expectation*—When we're talking about mock objects, an *expectation* is a feature built into the mock that verifies whether the external class calling this mock has the correct behavior. For example, a database connection mock could verify that the `close` method on the connection is called exactly once during any test that involves code using this mock.

For example, Figure 2 depicts a test of object A. To fulfil the needs of A, we discover that it needs a service S. While testing A we mock the responsibilities of S without defining a concrete implementation.

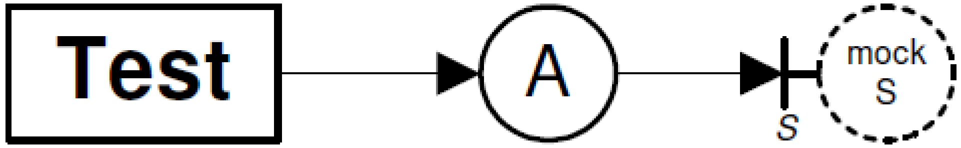


Figure 2. Interface Discovery

Once we have implemented A to satisfy its requirements we can switch focus and implement an object that performs the role of S. This is shown as object B in Figure 3. This process will then discover services required by B, which we again mock out until we have finished our implementation of B.

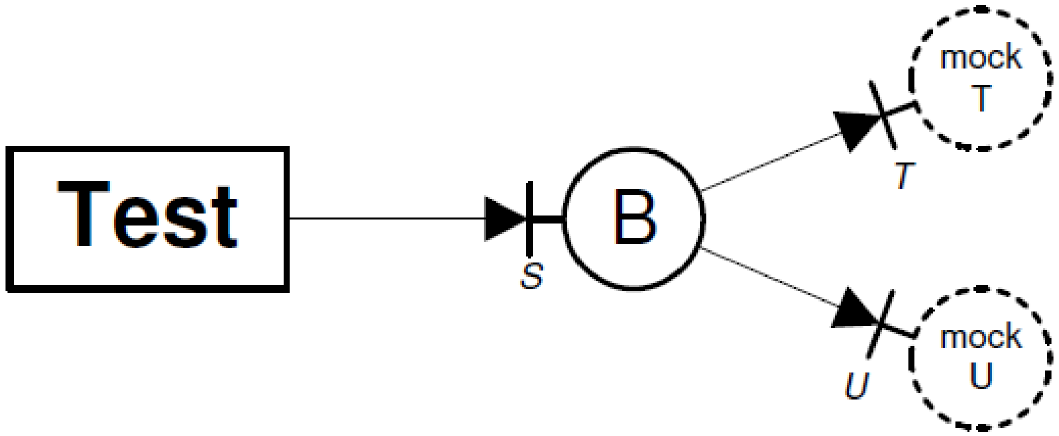
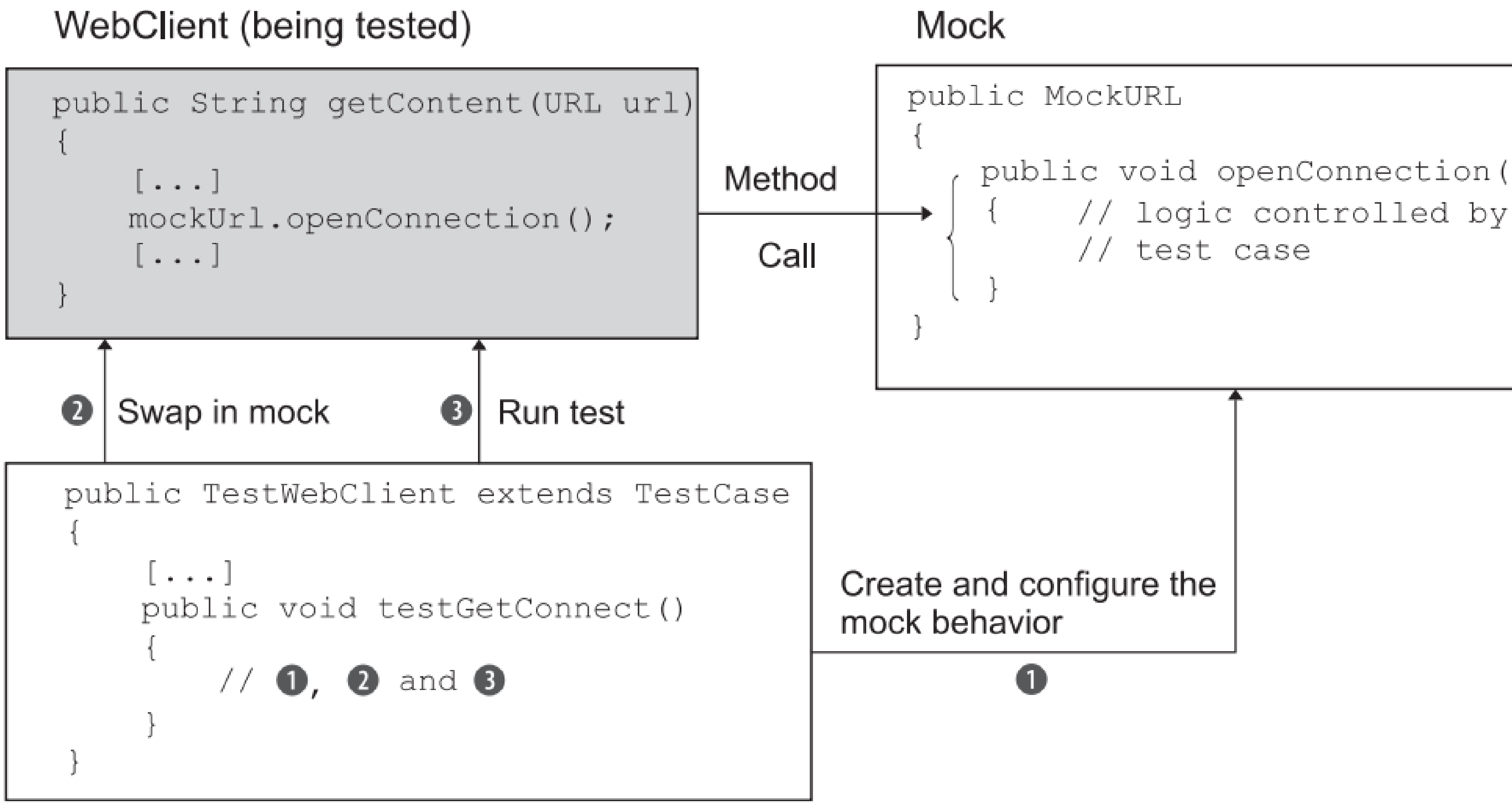


Figure 3. Iterative Interface Discovery

We continue this process until we reach a layer that implements real functionality in terms of the system runtime or external libraries.



```
public String getContent(URL url)
{
    [...]
    mockUrl.openConnection();
    [...]
}
```

```
public MockURL
{
    public void openConnection(
    {     // logic controlled by
          // test case
    }
}
```

```
public TestWebClient extends TestCase
{
    [...]
    public void testGetConnect()
    {
        // ❶, ❷ and ❸
    }
}
```

Figure 7.3 The steps involved in a test using mock objects

Stubs

Explicitly implements a simplified version of the target object behavior

Contains some business logic

Usually coarse

Mocks

Provides a generated object to respond to part of the target object's contract

No explicit implementation

Behavior specified by expectations

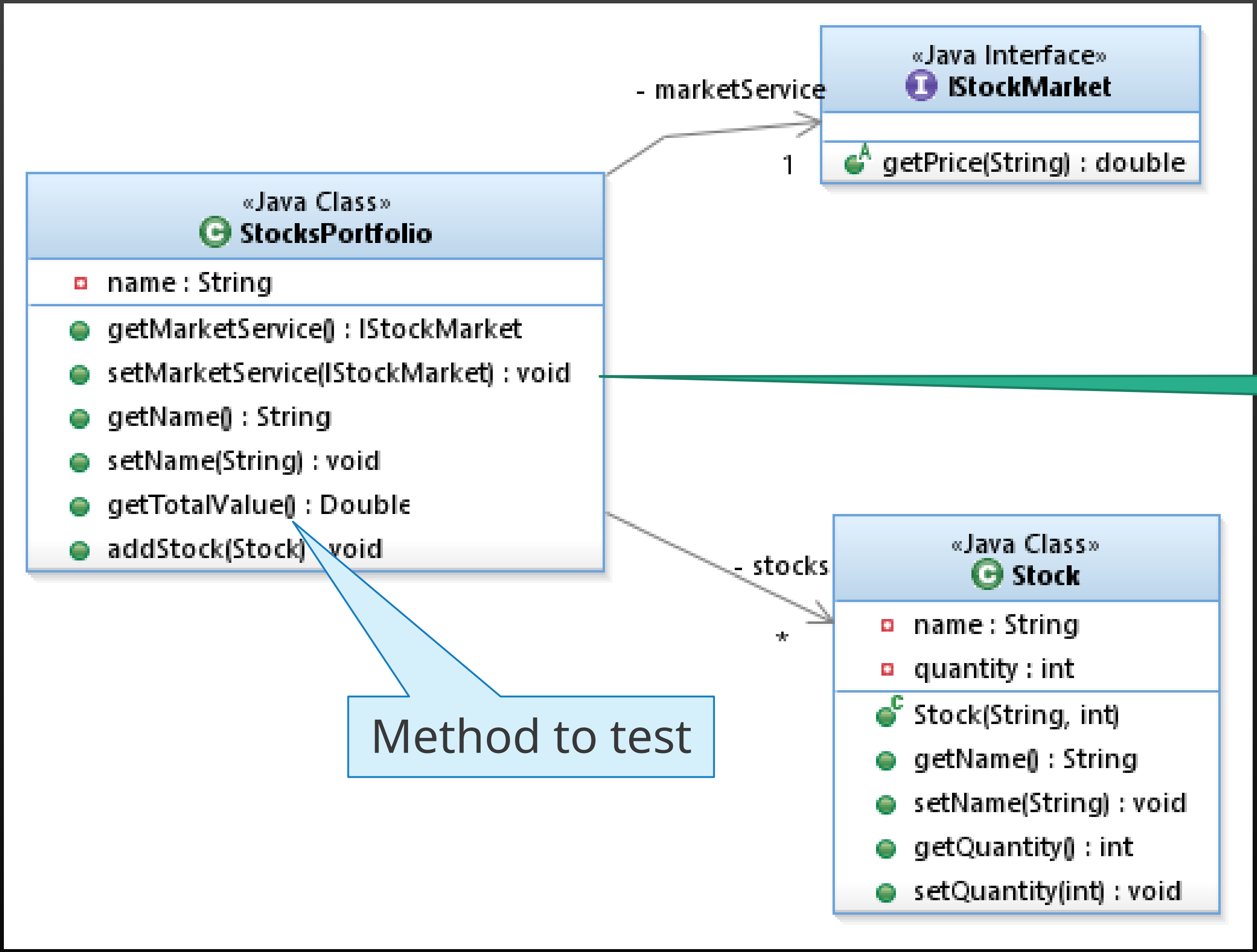
Fine-grained

method level

precise messages that pinpoint the cause of the breakage.

Example

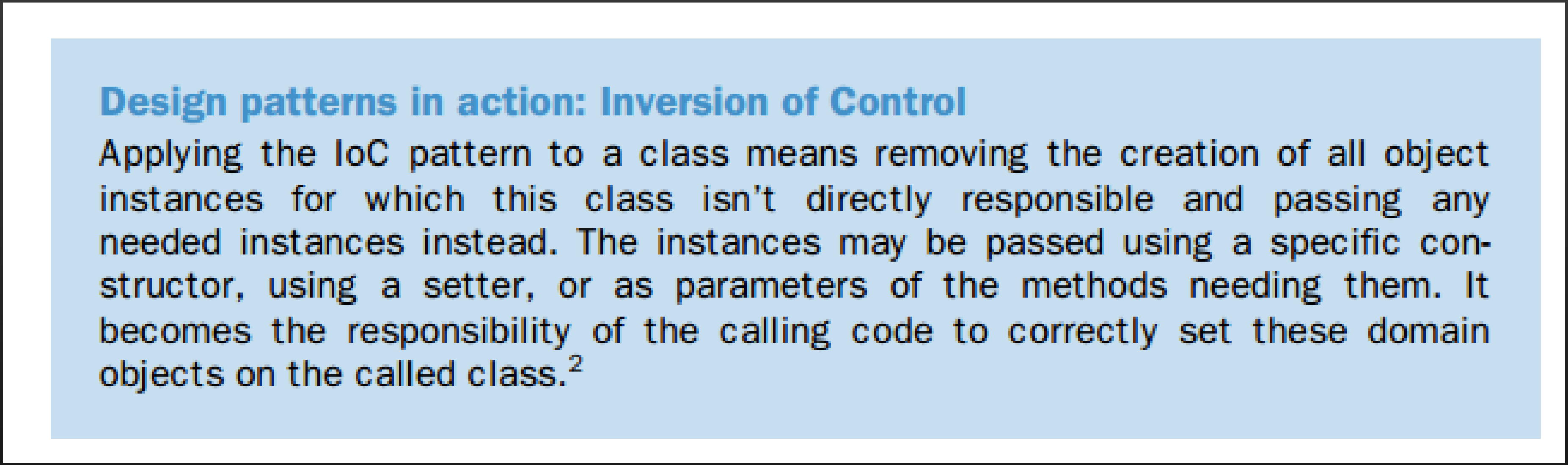
Easy mocking. Better testing.



Note the IoC

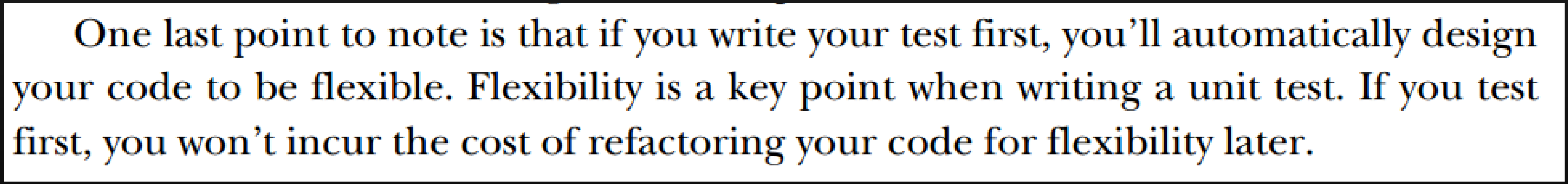
→ see also: DeveloperZone

Pattern to help with testing



Design patterns in action: Inversion of Control

Applying the IoC pattern to a class means removing the creation of all object instances for which this class isn't directly responsible and passing any needed instances instead. The instances may be passed using a specific constructor, using a setter, or as parameters of the methods needing them. It becomes the responsibility of the calling code to correctly set these domain objects on the called class.[2]

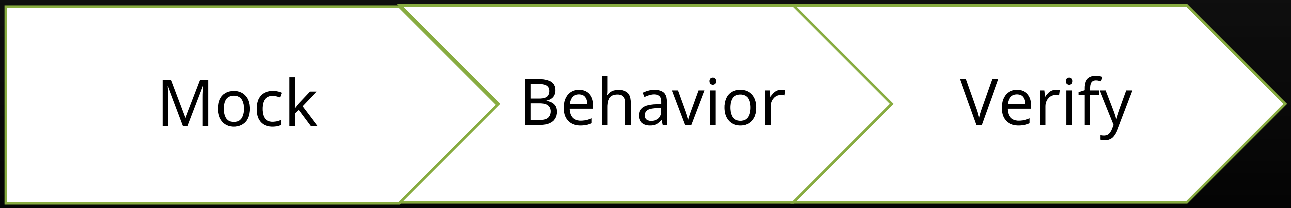


One last point to note is that if you write your test first, you'll automatically design your code to be flexible. Flexibility is a key point when writing a unit test. If you test first, you won't incur the cost of refactoring your code for flexibility later.

EasyMock workflow

http://easymock.org

1. create a mock object, given an interface
2. set expectations (specification of the expected calls)
   1. methods used, parameters, return values
   2. order (and number of times) of methods call
3. publish the specification (EasyMock.replay)
4. verify that expectations were met (EasyMock.verify)



EasyMock expectations

```
EasyMock.expect(mock.getPrice("EBAY")).andReturn(42.00);



EasyMock.expect(mock.getRate(
        (String) EasyMock.matches("[A-Z][A-Z][A-Z]"),
        (String) EasyMock.matches("[A-Z][A-Z][A-Z]"))).andReturn(1.5);



EasyMock.expect(mock.getRate("USD", "EUR")).andThrow(new
IOException());



EasyMock.expect(mock.getPrice("EBAY")).andReturn(42.00).times(3);
```

http://easymock.org/user-guide.html#verification-calls

EasyMock expectations – order of calls

Normal — EasyMock.createMock()

All of the expected methods must be called with the specified arguments; the order does not matter. Calls to unexpected methods cause test failure.

Strict — EasyMock.createStrictMock()

All expected methods must be called with the expected arguments, in a specified order. Calls to unexpected methods cause test failure.

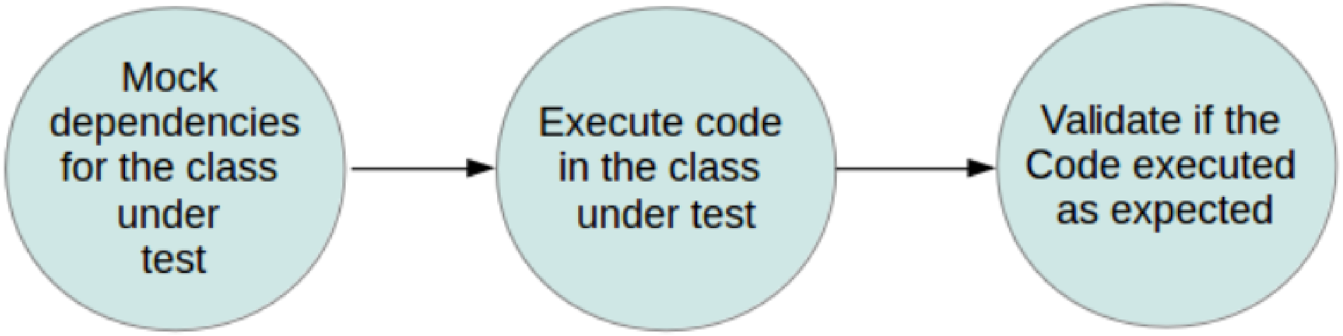
Nice — EasyMock.createNiceMock()

All expected methods must be called with the specified arguments in any order. Calls to unexpected methods do not cause the test to fail. Nice mocks supply reasonable defaults for methods you don't explicitly mock

http://easymock.org/user-guide.html#mocking-strict

If you use Mockito in tests you typically:

- Mock away external dependencies and insert the mocks into the code under test
- Execute the code under test
- Validate that the code executed correctly


Mock dependencies for the class under test
Execute code in the class under test
Validate if the Code executed as expected

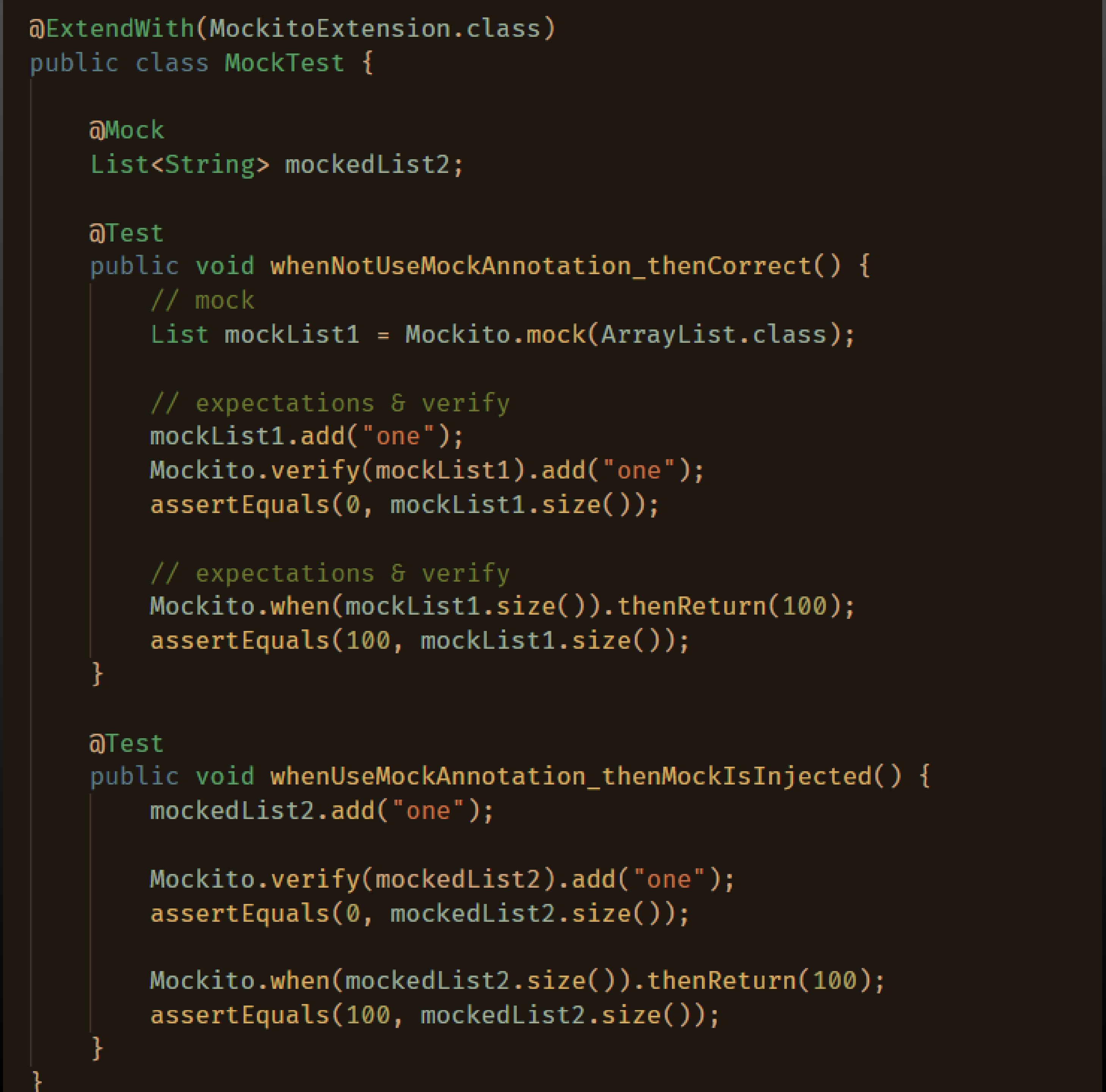
Mockito vocabulary

In mockito, we generally work with following kind of test doubles.

Stubs – is an object that has predefined return values to method executions made during the test.

Spies – are objects that are similar to stubs, but they additionally record how they were executed.

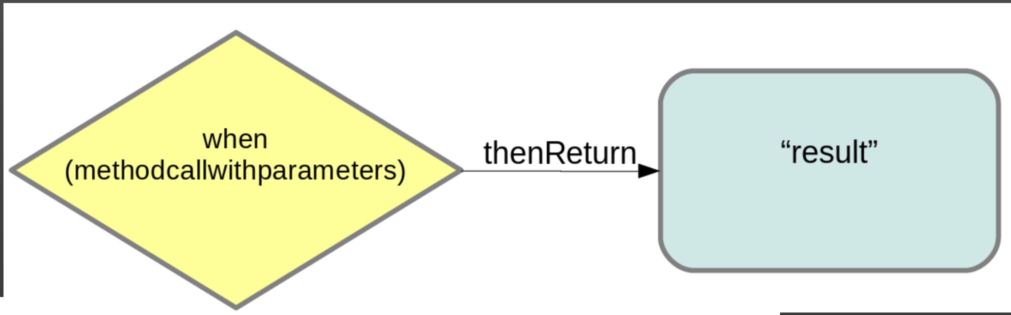
Mocks – are objects that have return values to method executions made during the test and has recorded expectations of these executions. Mocks can throw an exception if they receive a call they don’t expect and are checked during verification to ensure they got all the calls they were expecting.

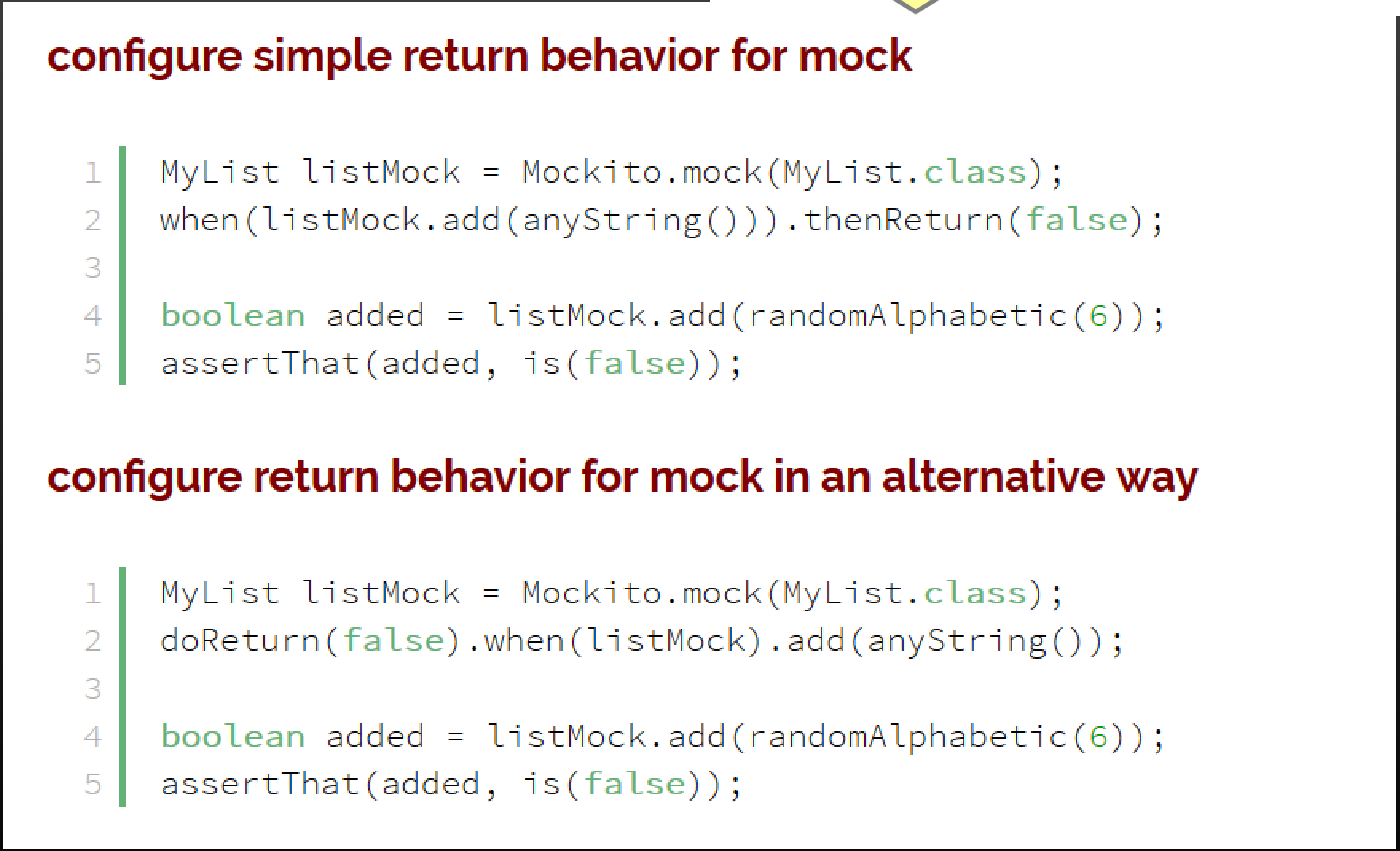


```
@ExtendWith(MockitoExtension.class)
public class MockTest {

    @Mock
    List<String> mockedList2;

    @Test
    public void whenNotUseMockAnnotation_thenCorrect() {
        // mock
        List mockList1 = Mockito.mock(ArrayList.class);

        // expectations & verify
        mockList1.add("one");
        Mockito.verify(mockList1).add("one");
        assertEquals(0, mockList1.size());

        // expectations & verify
        Mockito.when(mockList1.size()).thenReturn(100);
        assertEquals(100, mockList1.size());
    }

    @Test
    public void whenUseMockAnnotation_thenMockIsInjected() {
        mockedList2.add("one");

        Mockito.verify(mockedList2).add("one");
        assertEquals(0, mockedList2.size());

        Mockito.when(mockedList2.size()).thenReturn(100);
        assertEquals(100, mockedList2.size());
    }
}
```

I Oliveira (2020)

Simple expectations





configure simple return behavior for mock

```java
MyList listMock = Mockito.mock(MyList.class);
when(listMock.add(anyString())).thenReturn(false);

boolean added = listMock.add(randomAlphabetic(6));
assertThat(added, is(false));
```

configure return behavior for mock in an alternative way

```java
MyList listMock = Mockito.mock(MyList.class);
doReturn(false).when(listMock).add(anyString());

boolean added = listMock.add(randomAlphabetic(6));
assertThat(added, is(false));
```

https://www.baeldung.com/mockito-behavior

Expect exceptions from the mock

First, if our method return type is not *void* we can use *when().thenThrow()*:

```java
@Test(expected = NullPointerException.class)
public void whenConfigNonVoidRetunMethodToThrowEx_thenExIsThrown() {
    MyDictionary dictMock = mock(MyDictionary.class);
    when(dictMock.getMeaning(anyString()))
      .thenThrow(NullPointerException.class);

    dictMock.getMeaning("word");
}
```

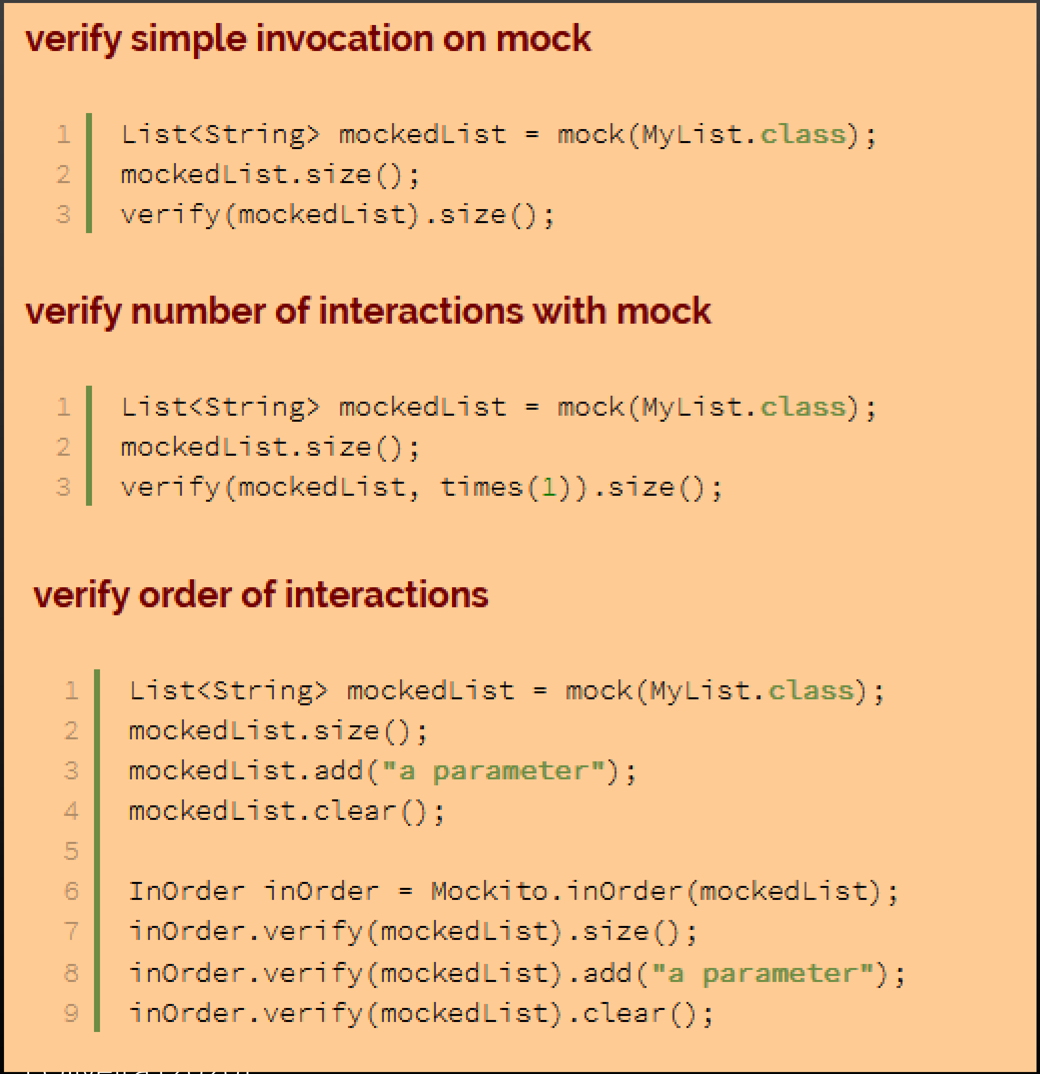
Notice, we configured the *getMeaning()* method – which returns a value of type *String* – to throw a *NullPointerException* when called.

Now, if our method returns *void*, we'll use *doThrow()*:

```java
@Test(expected = IllegalStateException.class)
public void whenConfigVoidRetunMethodToThrowEx_thenExIsThrown() {
    MyDictionary dictMock = mock(MyDictionary.class);
    doThrow(IllegalStateException.class)
      .when(dictMock)
      .add(anyString(), anyString());

    dictMock.add("word", "meaning");
}
```

I Oliveira (2020)

Mockito: verifying interactions



verify simple invocation on mock

```java
List<String> mockedList = mock(MyList.class);
mockedList.size();
verify(mockedList).size();
```

verify number of interactions with mock

```java
List<String> mockedList = mock(MyList.class);
mockedList.size();
verify(mockedList, times(1)).size();
```

verify order of interactions

```java
List<String> mockedList = mock(MyList.class);
mockedList.size();
mockedList.add("a parameter");
mockedList.clear();

InOrder inOrder = Mockito.inOrder(mockedList);
inOrder.verify(mockedList).size();
inOrder.verify(mockedList).add("a parameter");
inOrder.verify(mockedList).clear();
```

https://www.baeldung.com/mockito-verify

Annotations - creational

Annotation	Purpose
@Mock	create and **inject mocked instances** (without having to call Mockito.mock() “manually”) when()/given() to specify how a mock should behave
@Spy	**partial mocking**, real methods are invoked but still can be verified and stubbed. Every call, unless specified otherwise, is delegated to the object.
@Captor	Get the arguments used in a previous expectation.
@InjectMocks	Use the created mock and inject it as a field in test subject

When to use a mock object?

supplies non-deterministic results
e.g. the current time or the current temperature.

has states that are difficult to create or reproduce
e.g. a network error or database error.

is slow
e.g. a large network resource.

does not yet exist (test driven development) or may change behavior;

would have to include information and methods exclusively for testing purposes.

References

P. Tahchiev, F. Leme, V. Massol, and G. Gregory, JUnit in Action, Second Edition. Manning Publications, 2010.

Langr, J., Hunt, A. and Thomas, D., 2015. *Pragmatic Unit Testing in Java 8 with JUnit*. Pragmatic Bookshelf.

https://www.baeldung.com/mockito-series

https://github.com/mockito/mockito/wiki/How-to-write-good-tests