

45426: Teste e Qualidade de Software



# Unit tests and mock objects

Ilídio Oliveira

v2023-02-27

# 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?

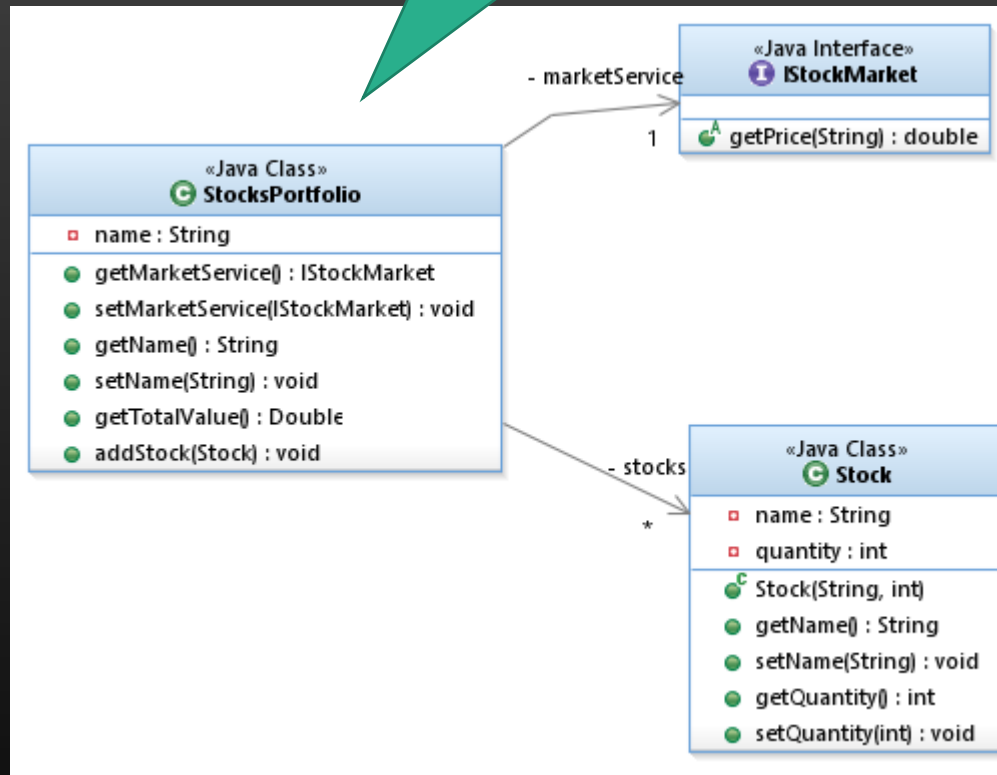
StockPortfolio queries StockMarket to update rates

To keep the isolation:

“fake” the behavior of the remote/real object/service

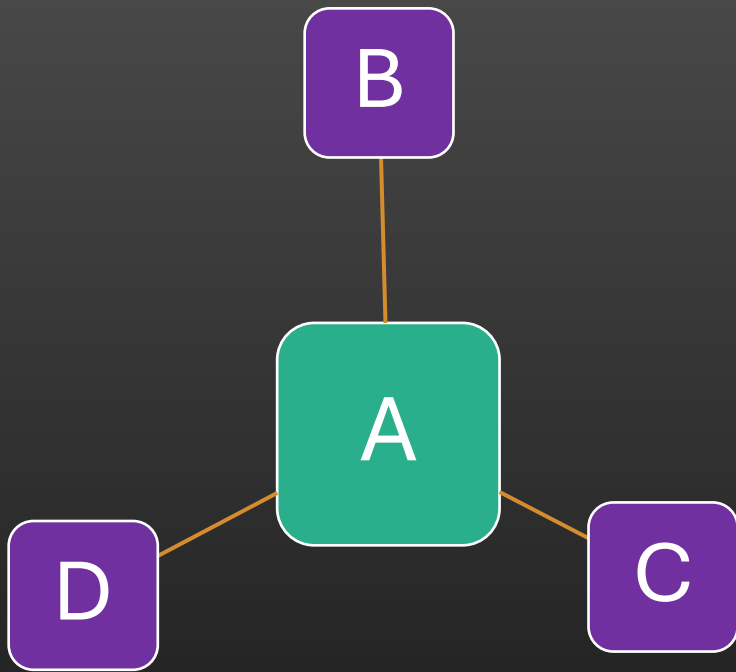
Just enough logic to get the tests done

Some methods of StockPortfolio depend on online services

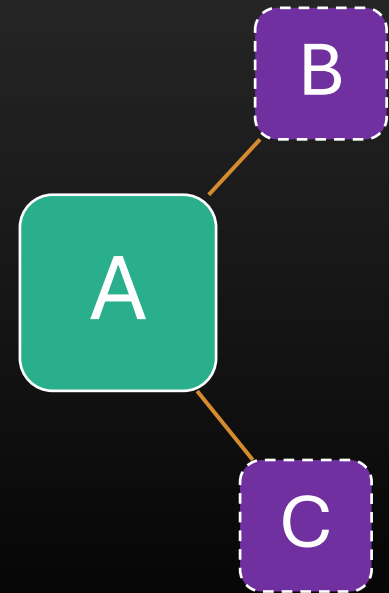


# Collaborators

## SuT (Subject under test)



## Test scenario



# 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

# [Manual] Stubs approach: drawbacks

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

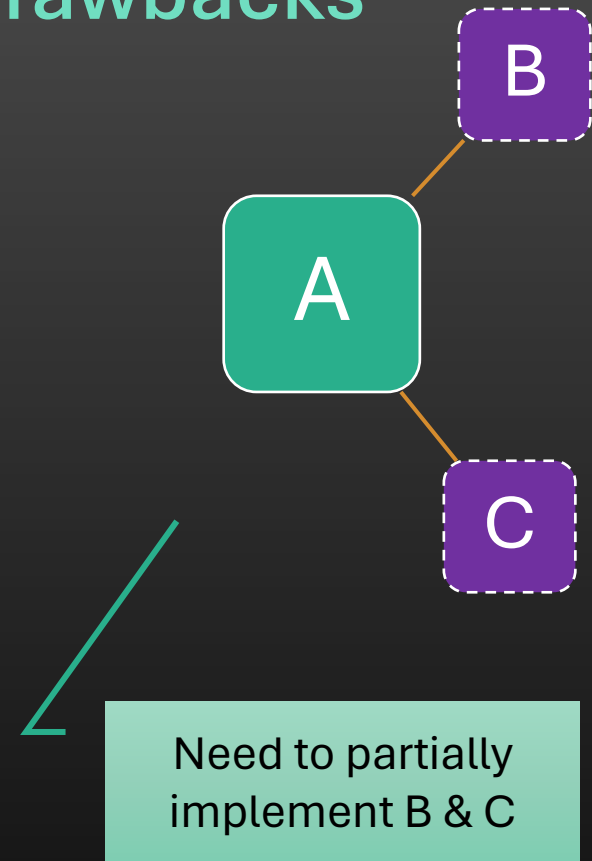
Often complex... stubs themselves need debugging!

Not 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.

# “Mocking” approach

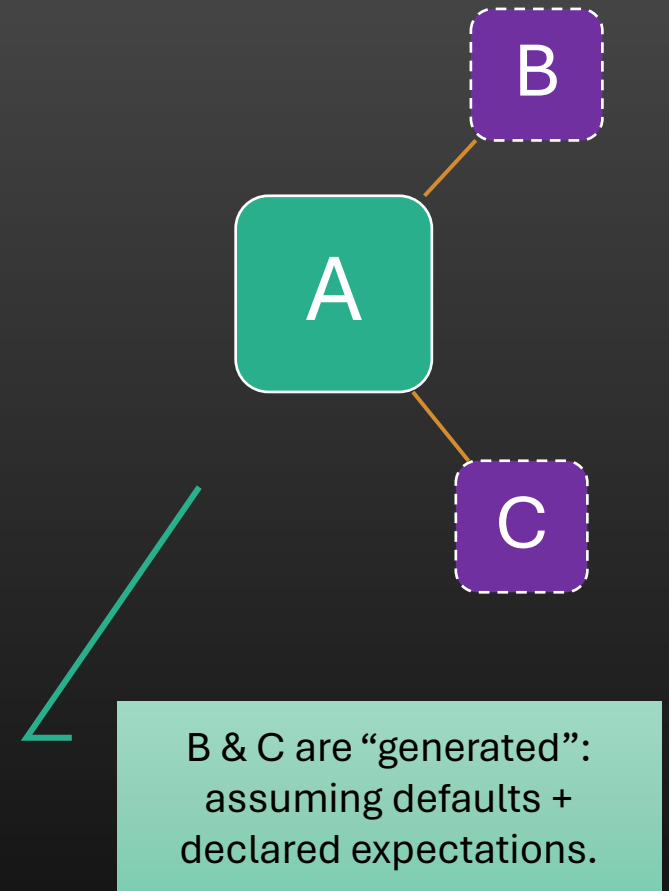
## The mock object is a test double

Allows the test case to describe the expected calls

The mock can check that, during the tests, all calls happened, with the right parameters, in the right order,...

The mock can be instructed to return specific values

Easy to “simulate” an expected scenario/behavior



**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.

## [Manual] 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

Automatic defaults (partially)

Behavior specified by expectations

Fine-grained

method level

precise messages that pinpoint the cause of the breakage.



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.

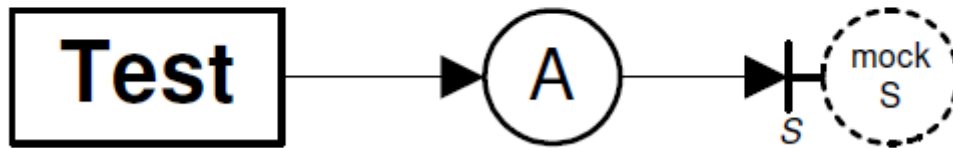


Figure 2. Interface Discovery

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.

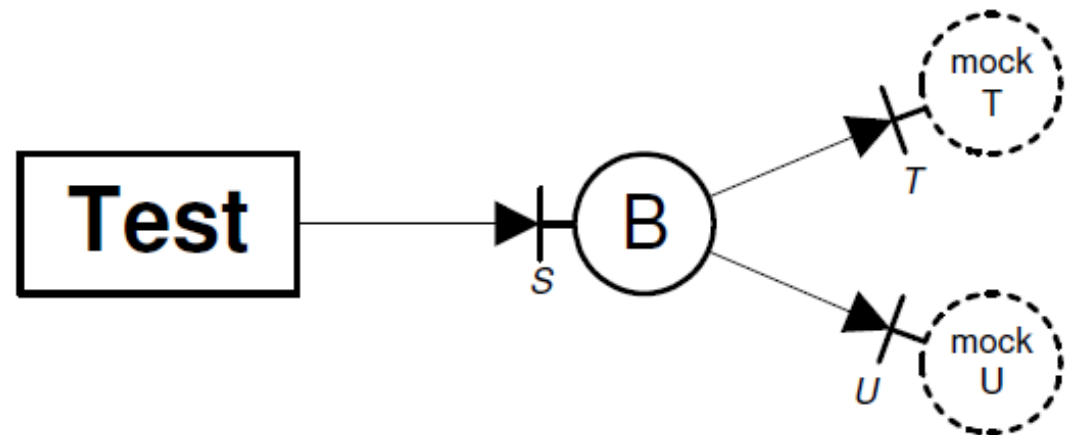
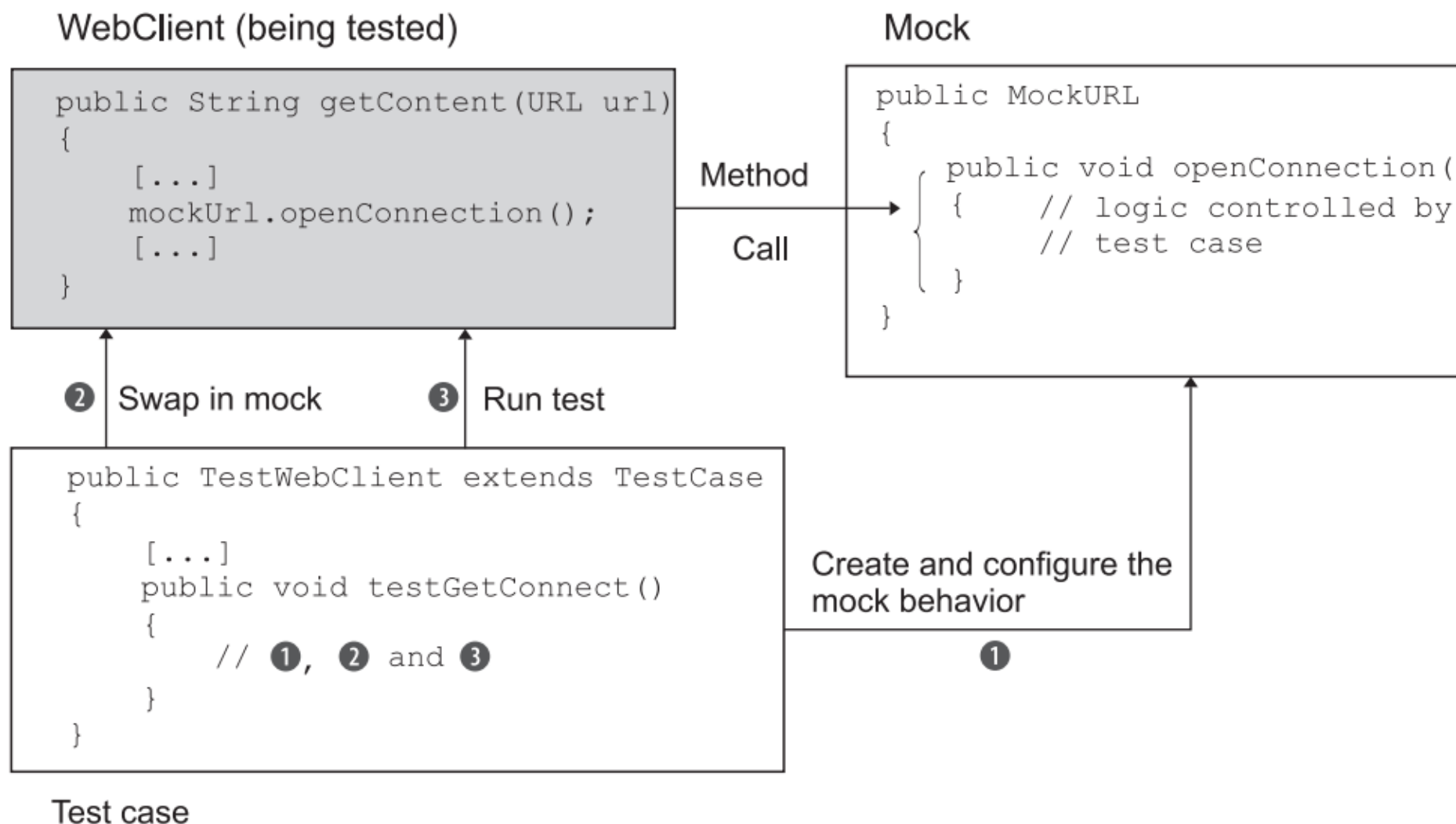


Figure 3. Iterative Interface Discovery

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

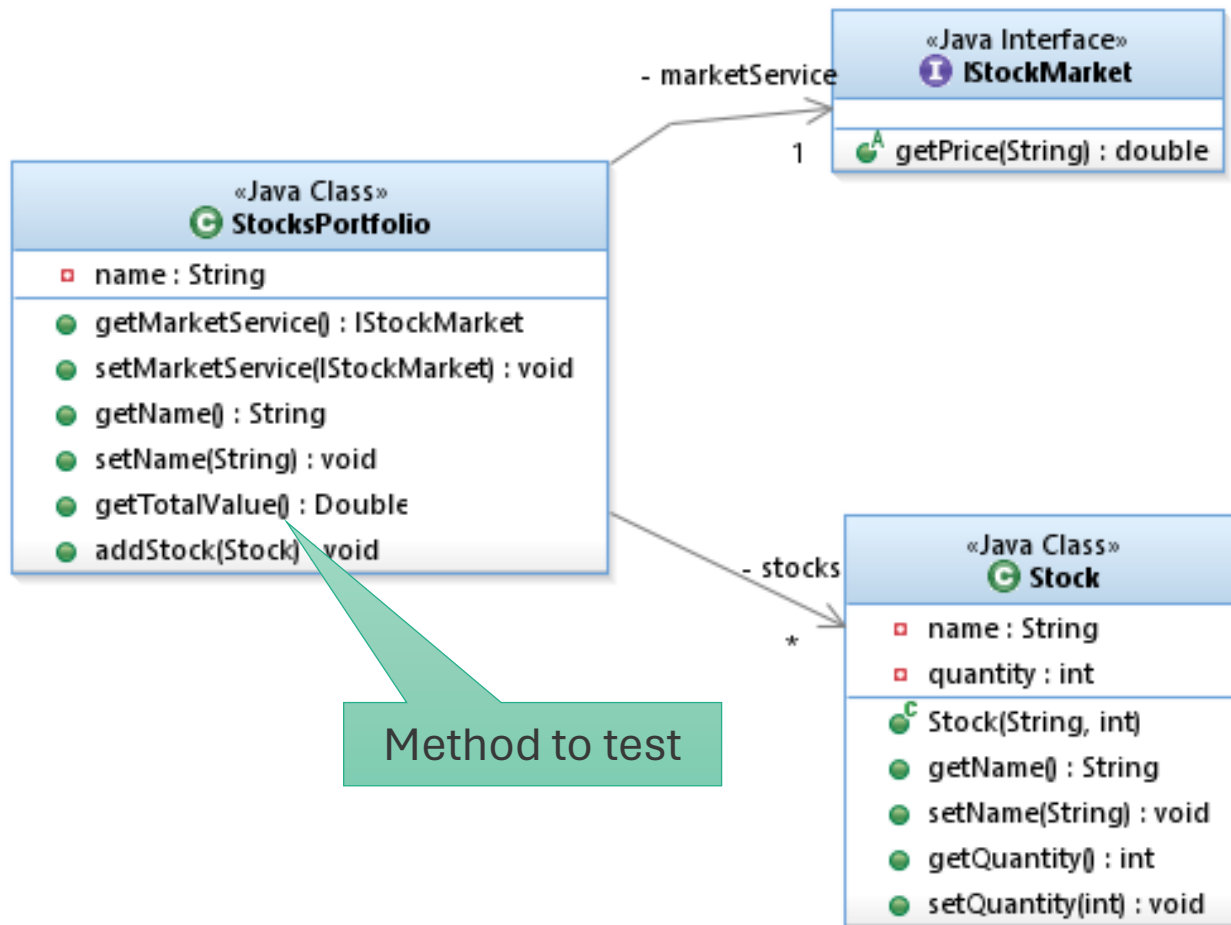


<https://learning.oreilly.com/library/view/growing-object-oriented-software/9780321574442/>



**Figure 7.3** The steps involved in a test using mock objects

# Example



# Pattern to help with testing

## Design patterns in action: Inversion of Control

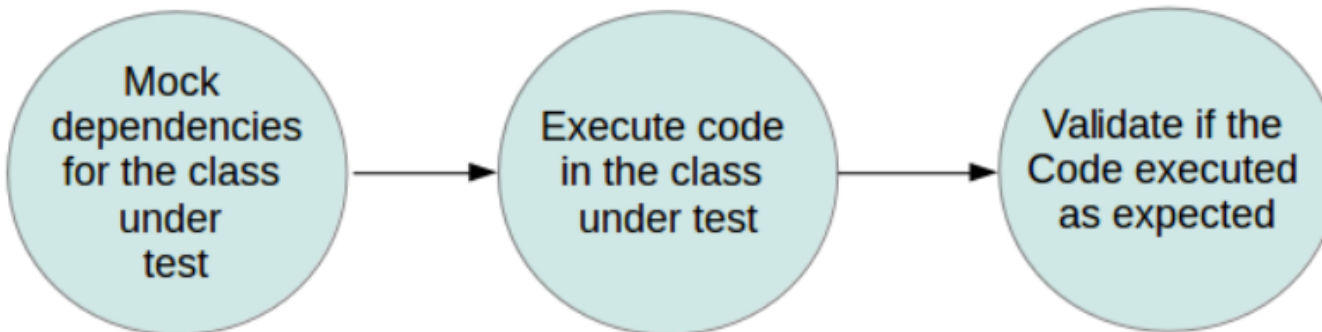
Applying the IoC pattern to a class means removing the creation of all object instances for which this class isn't directly responsible and passing any needed instances instead. The instances may be passed using a specific constructor, using a setter, or as parameters of the methods needing them. It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

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.



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



now you can verify interactions

Inspect if/how the  
operations of interest were  
used.  
(Test interactions, not the  
return values.)

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

// mock creation
List mockedList = mock(List.class);

// using mock object - it does not throw any "unexpected interaction" exc
mockedList.add("one");
mockedList.clear();

// selective, explicit, highly readable verification
verify(mockedList).add("one");
verify(mockedList).clear();
```

Prepare canned responses.

and stub method calls

```
// you can mock concrete classes, not only interfaces
LinkedList mockedList = mock(LinkedList.class);

// stubbing appears before the actual execution
when(mockedList.get(0)).thenReturn("first");

// the following prints "first"
System.out.println(mockedList.get(0));

// the following prints "null" because get(999) was not stubbed
System.out.println(mockedList.get(999));
```

```
@Test
void whenGetTotal_thenSumWithMockedMarket() {

    // 1- instantiate the mock substitute
    IStockmarketService market = Mockito.mock( IStockmarketService.class );

    // 2- instantiate the SuT and inject the mock
    StocksPortfolio portfolio = new StocksPortfolio(market);

    // 3- "teach" the required expectations ("charge" the mock)
    Mockito.when( market.lookupPrice("EBAY")).thenReturn( 4.0);
    Mockito.when( market.lookupPrice("MSFT")).thenReturn( 1.5);

    // 4- execute the test in the SuT
    portfolio.addStock(new Stock("EBAY", 2));
    portfolio.addStock(new Stock("MSFT", 4));
    double result = portfolio.getTotalValue();

    // 5- verify the result and/or the use of the mock object
    assertEquals( 14.0, result);
    Mockito.verify( market, Mockito.times(2)).lookupPrice( Mockito.anyString() );
}
```

```
@ExtendWith(MockitoExtension.class) // necessário para usar anotações
```

```
class StocksPortfolio3Test {
```

```
    @Mock
```

```
    IStockmarketService market;
```

```
    @InjectMocks
```

```
    StocksPortfolio portfolio;
```

```
    @Test
```

```
    void getTotalValueAnnot() {
```

```
        // 1. Prepare a mock to substitute the remote service (@Mock annotation)
```

```
        // 2. Create an instance of the subject under test (SuT) and use the mock to set the
```

```
        // 3. Load the mock with the proper expectations (when...thenReturn)
```

```
        when( market.getPrice( stockLabel: "EBAY" ) ).thenReturn( 4.0 );
```

```
        when( market.getPrice( stockLabel: "MSFT" ) ).thenReturn( 1.5 );
```

```
        // when( market.getPrice( "NOTUSED" ) ).thenReturn( 1.5 );
```

```
        // 4. Execute the test (use the service in the SuT)
```

```
        portfolio.addStock( new Stock( label: "EBAY", quantity: 2 ) );
```

```
        portfolio.addStock( new Stock( label: "MSFT", quantity: 4 ) );
```

```
        double result = portfolio.getTotalValue();
```

```
        // 5. Verify the result (assert) and the use of the mock (verify)
```

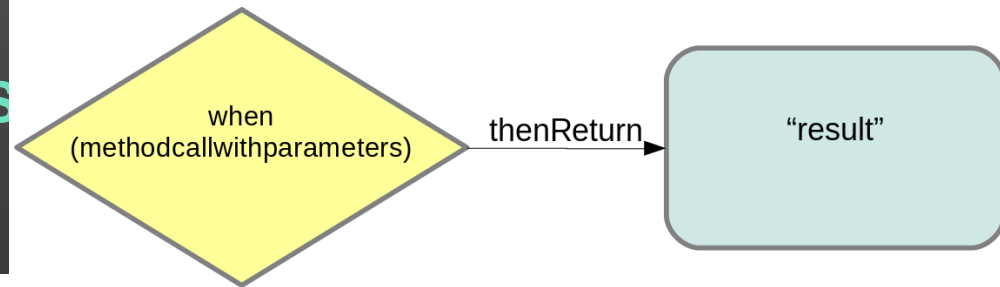
```
        assertEquals( expected: 14.0, result );
```

```
        verify( market, times( wantedNumberOfInvocations: 2 ) ).getPrice( anyString() );
```

```
    }
```



# Simple expectations



## configure simple return behavior for mock

```
1 | MyList listMock = Mockito.mock(MyList.class);
2 | when(listMock.add(anyString())).thenReturn(false);
3 |
4 | boolean added = listMock.add(randomAlphabetic(6));
5 | assertThat(added, is(false));
```

## configure return behavior for mock in an alternative way

```
1 | MyList listMock = Mockito.mock(MyList.class);
2 | doReturn(false).when(listMock).add(anyString());
3 |
4 | boolean added = listMock.add(randomAlphabetic(6));
5 | assertThat(added, is(false));
```

# Expect exceptions from the mock

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

```
1  @Test(expected = NullPointerException.class)
2  public void whenConfigNonVoidRetunMethodToThrowEx_thenExIsThrown() {
3      MyDictionary dictMock = mock(MyDictionary.class);
4      when(dictMock.getMeaning(anyString()))
5          .thenThrow(NullPointerException.class);
6
7      dictMock.getMeaning("word");
8  }
```

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()*.

```
1  @Test(expected = IllegalStateException.class)
2  public void whenConfigVoidRetunMethodToThrowEx_thenExIsThrown() {
3      MyDictionary dictMock = mock(MyDictionary.class);
4      doThrow(IllegalStateException.class)
5          .when(dictMock)
6          .add(anyString(), anyString());
7
8      dictMock.add("word", "meaning");
9  }
```

# Mockito: verifying interactions (not only results)

## verify simple invocation on mock

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

## verify number of interactions with mock

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

## verify order of interactions

```
1 List<String> mockedList = mock(MyList.class);
2 mockedList.size();
3 mockedList.add("a parameter");
4 mockedList.clear();
5
6 InOrder inOrder = Mockito.inOrder(mockedList);
7 inOrder.verify(mockedList).size();
8 inOrder.verify(mockedList).add("a parameter");
9 inOrder.verify(mockedList).clear();
```

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

# Example

```
// You can mock concrete classes and interfaces
TrainSeats seats = mock(TrainSeats.class);

// stubbing appears before the actual execution
when(seats.book(Seat.near(WINDOW).in(FIRST_CLASS))).thenReturn(BOOKED);

// the following prints "BOOKED"
System.out.println(seats.book(Seat.near(WINDOW).in(FIRST_CLASS)));

// the following prints "null" because
// .book(Seat.near(AISLE).in(FIRST_CLASS)) was not stubbed
System.out.println(seats.book(Seat.near(AISLE).in(FIRST_CLASS)));

// the following verification passes because
// .book(Seat.near(WINDOW).in(FIRST_CLASS)) has been invoked
verify(seats).book(Seat.near(WINDOW).in(FIRST_CLASS));

// the following verification fails because
// .book(Seat.in(SECOND_CLASS)) has not been invoked
verify(seats).book(Seat.in(SECOND_CLASS));
```

# Spying

Note the test method naming style:  
when**Action**\_then**Expected**

```
@Spy
List<String> spiedList = new ArrayList<String>();

@Test
public void whenUseSpyAnnotation_thenSpyIsInjectedCorrectly() {
    spiedList.add("one");
    spiedList.add("two");

    Mockito.verify(spiedList).add("one");
    Mockito.verify(spiedList).add("two");

    assertEquals(2, spiedList.size());

    Mockito.doReturn(100).when(spiedList).size();
    assertEquals(100, spiedList.size());
}
```

# Captor

```
@Mock
List mockedList;

@Captor
ArgumentCaptor argCaptor;

@Test
public void whenUseCaptorAnnotation_thenTheSam() {
    mockedList.add("one");
    Mockito.verify(mockedList).add(argCaptor.capture());

    assertEquals("one", argCaptor.getValue());
}
```

# Mockito Annotations (requires enabling)

Annotation	Purpose
@Mock	create <b>mocked instances</b> (without having to call Mockito.mock() “manually”) <a href="#">when()</a> to specify how a mock should behave
@Spy	<b>partial mocking</b> , 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 currency exchange-rate or the current temperature.

## **has states that are difficult to create or reproduce**

e.g. a specific network error or database error.

## **is slow**


e.g. a network resource (latency).

## **module does not yet exist**

e.g.: TDD, a module to be later implemented → would have to include information and methods exclusively for testing purposes.

## **But do not use for...**

- Data/value classes
- “Everywhere” ....
- Careful about API/services you don’t control



If a mocked (3<sup>rd</sup> party) API changes, what happens?

- to the test?
- to the production code?



# Best practices

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

## 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://site.mockito.org/>

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

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