

Extracted from:

Mockito Made Clear

Java Unit Testing with Mocks, Stubs, and Spies

This PDF file contains pages extracted from *Mockito Made Clear*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit <http://www.pragprog.com>.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2023 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Raleigh, North Carolina

The
Pragmatic
Programmers

Mockito Made Clear

*Java Unit Testing
with Mocks, Stubs,
and Spies*



Ken Kousen

Foreword by Venkat Subramaniam

Edited by Margaret Eldridge

Mockito Made Clear

Java Unit Testing with Mocks, Stubs, and Spies

Ken Kousen

The Pragmatic Bookshelf

Raleigh, North Carolina



Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

For our complete catalog of hands-on, practical, and Pragmatic content for software developers, please visit <https://pragprog.com>.

The team that produced this book includes:

CEO: Dave Rankin

COO: Janet Furlow

Managing Editor: Tammy Coron

Development Editor: Margaret Eldridge

Founders: Andy Hunt and Dave Thomas

For sales, volume licensing, and support, please contact support@pragprog.com.

For international rights, please contact rights@pragprog.com.

Copyright © 2023 The Pragmatic Programmers, LLC.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

ISBN-13: 978-1-68050-967-0

Encoded using the finest acid-free high-entropy binary digits.

Book version: P1.0—January 2023

*To Ginger and Xander, the people that matter
most to me in the world.*

Using Mocks and Stubs in the Astro Project

In [Counting Astronauts by Spaceship, on page ?](#), we introduced a project that processed the number of astronauts in space and returned how many were aboard each spacecraft. The examples there showed an integration test for the `AstroService` class and how to create a unit test using our own `FakeGateway`. Now we can use Mockito instead to mock the `Gateway<T>` interface.

As a reminder, `AstroService` retrieves the data from a `Gateway<T>`, reproduced here:

Gateway.java

```
public interface Gateway<T> {  
    T getResponse();  
}
```

The class that implements this interface is called `AstroGateway`, and it returns an `AstroResponse` that could possibly be null. If we're going to isolate the `AstroService`, we need to ask Mockito to return either that `AstroResponse` or throw a `RuntimeException` if it's null.

Let's use JUnit 5 along with the Mockito annotations. The test class for the `AstroService` now starts with this:

AstroServiceTest.java

```
@ExtendWith(MockitoExtension.class)  
class AstroServiceTest {  
  
    @Mock  
    private Gateway<AstroResponse> gateway;  
  
    @InjectMocks  
    private AstroService service;
```

We'll rely on Mockito to create the mock of the gateway and then inject it into the service because we've provided a constructor to the `AstroService` class that takes a `Gateway` as an argument.

One test then looks like this:

AstroServiceTest.java

```
@Test  
void testAstroData_usingInjectedMockGateway() {  
    // Mock Gateway created and injected into AstroService using  
    //    @Mock and @InjectMock annotations  
    //  
    // Set the expectations on the mock  
    when(gateway.getResponse())  
        .thenReturn(mockAstroResponse);  
  
    // Call the method under test
```



```

Map<String, Long> astroData = service.getAstroData();
// Check the results from the method under test
assertThat(astroData)
    .containsEntry("Babylon 5", 2L)
    .containsEntry("Nostramo", 1L)
    .containsEntry("USS Cerritos", 4L);
astroData.forEach((craft, number) -> {
    System.out.println(number + " astronauts aboard " + craft);
    assertAll(
        () -> assertThat(number).isPositive(),
        () -> assertThat(craft).isNotBlank()
    );
});
// Verify the stubbed method was called
verify(gateway).getResponse();
}

```

The annotations take care of two of our steps: creating the mock and injecting into the service. We still need to set the expectations, which we can do with the `when/thenReturn` pair. After calling the method under test `getAstroData` in `AstroService`, we can then verify that the results are correct. At the end, we can optionally check that the service did indeed invoke the `getResponse` method on the gateway exactly once.

Testing for a failure is similar and uses the `thenThrow` method:

```

AstroServiceTest.java
// Check network failure
@Test
void testAstroData_usingFailedGateway() {
    when(gateway.getResponse()).thenThrow(
        new RuntimeException(new IOException("Network problems")));
    assertThatExceptionOfType(RuntimeException.class)
        .isThrownBy(() -> service.getAstroData())
        .withCauseInstanceOf(IOException.class)
        .withMessageContaining("Network problems");
}

```

Since the `getResponse` method on the gateway returns a `Failure` that wraps the exception, we don't have to use the `doThrow/when` construct, which is required when the method we're mocking returns `void`.

Another advantage of using Mockito is you can test for the case of network failure without disabling the network. I bet the astronauts would be happy about that.

Congratulations. You've successfully used Mockito to mock the Gateway, and that concludes round one of the Mockito API game.

Wrapping Up

You can either create your mocks manually and insert them into your class under test or you can use Mockito's annotation support to inject them automatically. Mockito doesn't provide a full injection framework, like Spring or Guice, but it does make a good faith effort to plug in your mocks via either constructor injection, setter injection, or even field injection.

You now have the mechanisms for telling your mocks (or stubs, in this case) what to return when methods are called. Chained methods made the process much simpler, and you can use multiple arguments to respond to each invocation with different results. You also now know about the special handling required to mock methods that return void.

In the next chapter, we'll talk about generalizing the arguments to mocked methods so you don't have to give explicit values every time. Mockito has a class that contains factory methods for that purpose—we'll also use Java lambdas to implement custom argument matchers.