Extracted from:

Mockito Made Clear

Java Unit Testing with Mocks, Stubs, and Spies

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

Pragmatic Bookshelf

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*.

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.

Counting Astronauts by Spaceship

How many astronauts are on each ship in space right now? Let's talk through a modern Java solution that doesn't require many classes but uses a familiar architecture you'll find in many applications. We'll use this problem as an introduction to testing in general and Mockito in particular.

The final product will have three components:

1. A class called `AstroGateway` that accesses a RESTful web service to retrieve the astronaut data in JSON form and then converts the JSON structure to Java POJOs (plain old Java objects).
2. A class called `AstroService` that processes the Java POJOs and returns a `Map` of strings to integers, where the map keys are the spacecraft names and the map values are the number of astronauts aboard each.
3. An application class that drives the process, which in our case will be a series of test cases that invoke the right methods and print and verify the results.

A Note on Java Versions

Mockito requires only Java 8. The current Long Term Support (LTS) version of Java is 17, but as of 2022, the community is split between Java 8 and the previous LTS version, Java 11.

The implementation of this "astro" system uses the HTTP client added to Java in version 11, and 11 is the default version of Java we'll use throughout the book. None of the code uses records, sealed classes, pattern matching, switch expressions, or any of the other newer features added since 11.

For those people still on Java 8, first, you have my condolences, and second, I've included an alternative implementation of the gateway that uses the Retrofit library. Hopefully, this way, everybody will feel included.

We'll be testing an existing code implementation that we'll discuss as we go along. If you're curious about the details and want to see them now, take a look at the book's GitHub repository.[3] The relevant code is in the `com.kousenit.astro` package.

3. https://github.com/kousen/mockitobook

As a reminder, to add Mockito to a project, the only dependencies required are mockito-core and, when using the Mockito JUnit 5 extension, as here, mockito-junit-jupiter.

With the project configured for testing, let's add the classes to do the job.

Creating the Basic Classes

The astronaut data comes from the free RESTful web service at Open Notify[4] called People in Space.[5] It supports only HTTP GET requests, but it's free and doesn't require registration or any kind of key. It returns a JSON response showing all the astronauts in space at any given moment. If you send an HTTP GET request to http://api.open-notify.org/astros.json, you'll get back a response similar to this:

astro_data.json
```json
{
  "message": "success",
  "people": [
    {
      "name": "Bob Hines",
      "craft": "ISS"
    },
    {
      "name": "Oleg Artemyev",
      "craft": "ISS"
    },
    ...,
    {
      "name": "Cai Xuzhe",
      "craft": "Tiangong"
    }
  ],
  "number": 10
}
```

As you can see from the sample, the response includes the total number of astronauts and a collection called people which contains name and craft combinations for each astronaut. Java POJOs make a convenient structure to hold the parsed JSON information. Only two Java classes are needed:

- Assignment, representing the astronaut name and craft pair, and
- AstroResponse, which shows the total number of people in space, a success message, and the people array or list of the assignments.

4. https://open-notify.org
5. http://open-notify.org/Open-Notify-API/People-In-Space/

The Java classes will be implemented as traditional POJOs, meaning they'll have attributes that match the JSON properties, along with getter and setter methods. Here's the Assignment class:

Assignment.java

```java
public class Assignment {
    private final String name;
    private final String craft;

    // constructors, getters and setters, toString
}
```

Here's the AstroResponse class:

AstroResponse.java

```java
public class AstroResponse {
    private final int number;
    private final String message;
    private final List<Assignment> people;

    // constructors, getters and setters, toString
}
```

To access the RESTful web service, we'll use the Gateway[6] design pattern. A gateway is a class that encapsulates access to an external resource. In this particular case, we need only a single gateway, called an AstroGateway, that connects to the remote service, downloads the astro data, and converts the JSON response into records. Java is happiest when you use interfaces, and doing so here will make testing easier because we'll be able to substitute in a fake gateway when we need one. Therefore, add a Gateway interface that contains a single method called getResponse:

Gateway.java

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

The getResponse method returns an instance of its generic type.

The concrete class implementing the Gateway interface will be AstroGateway. You'll implement it using either the HttpClient API if you're using Java 11+, or the Retrofit 2[7] library if you're still on Java 8.

6. https://martinfowler.com/articles/gateway-pattern.html
7. https://square.github.io/retrofit/

Abstraction Theater

Introducing an interface (Gateway) with only a single implementation (AstroGateway) may feel like an unnecessary distraction. In this particular case, there will be two implementations: one that uses the new HttpClient API added in Java 11 and one that uses the external Retrofit library, which you can use if Java 11 isn't available. Only the HttpClient version will be included here, but both are contained in the GitHub repository for the book.

We'll get to the tests next in Adding Unit and Integration (End-to-End) Tests, on page ?. Here we're defining the basic classes, so let's also define the AstroService class, whose job is to convert the records returned by the gateway into a Java Map. The class needs only a single method, called getAstroData, with the following signature:

```
public Map<String,Long> getAstroData()
```

The AstroService class uses the gateway to access the remote web service. The gateway retrieves the remote data and turns it into either a Success instance containing an AstroResponse or a Failure instance containing any thrown exception if not. Assuming it's successful, the service then extracts the data we want and converts it to a map, where the keys are spacecraft names and the values are the number of astronauts aboard each one.

In the end, our application instantiates the AstroService, calls its getAstroData method, and processes the results. The results (based on the JSON data shown earlier) will be as follows:

```
3 astronauts aboard Tiangong
7 astronauts aboard ISS
```

Now let's turn to the tests necessary to verify the implementation works as desired.