Desarrollo de software

Usando Mockito

Entregable: Presenta en tu repositorio una carpeta llamada Mockito con tus respuestas

Una nota sobre las versiones de software: la versión de Mockito usada para esta actividad es 4.0.0. Es posible que esté utilizando una versión de la línea 2 o de la línea 3. Está bien, porque estas son las únicas diferencias:

- Mockito 3 es solo Mockito 2 con una versión Java requerida de 1.8 o superior.
- Mockito 4 es solo Mockito 3 con elementos obsoletos eliminados.
- Mockito 5 es solo Mockito 4 con el creador de mocks en línea incluido en el core

En otras palabras, todo el código de esta actividad está escrito para Mockito 4 pero debería funcionar en cualquier versión de Mockito 2.* o superior. La versión de Java será la 11 porque aunque la versión LTS (Long Term Support) actual es la 17, no muchos usuarios de la comunidad se han pasado a esa versión. **Todo el código del curso funcionará en todas las versiones de Java desde la 1.8 hasta la 19.**

Todas las pruebas se basan en el marco de pruebas JUnit.

Ahora es el momento de empezar a analizar los tipos de problemas para los que Mockito está diseñado.

Parte 1

Construye una base de pruebas

Mockito es una herramienta que te ayuda a aislar componentes particulares de software. Utiliza Mockito para reemplazar las dependencias de los componentes que estás probando, de modo que los métodos en cada dependencia devuelvan salidas conocidas para entradas conocidas. De esa manera, si ocurre un error al probar el componente, sabrá dónde y por qué.

Un desafío al usar Mockito es que se basa en una base de principios de prueba que es necesario conocer para usar la herramienta correctamente. En esta parte usaremos un ejemplo interesante para ilustrar esos fundamentos para que luego podamos centrarnos en cómo hacer que Mockito haga lo que tu quieres que haga.

Saludando a Mockito

Desde la publicación original de The C Programming Language[2] por Kernighan y Richie (1978), cada lenguaje y marco de programación debe, por ley, incluir un ¡Hola, mundo! Veremos la versión de Mockito en esta sección, explicaremos qué hace Mockito y para qué sirve, e introduciremos parte de un sistema más amplio que usaremos para discusiones posteriores más adelante en el curso.

IMPORTANTE: Construye un proyecto Gradle, con estos archivos.

Para Java, el clásico ¡Hola mundo! El programa es este, por supuesto:

HelloWorld.java

```
public class HelloWorld {

    public static void main(String[] args) {

        System.out.println("Hello, World!");

    }

}
```

La clase HelloWorld contiene un método main, pero main devuelve void, y los métodos void son difíciles de probar. Mockito es una biblioteca de prueba, así que modifiquemos esto para crear una clase que dé la bienvenida al usuario con una cadena simple devuelta por un método de bienvenida:

Eso es mejor. Es bastante fácil crear un caso de prueba JUnit 5 para el método greet:

HelloMockitoTest.java

```
import org.junit.jupiter.api.Test;

class HelloMockitoTest {

    private HelloMockito helloMockito = new HelloMockito();

    @Test

    void greetPerson() {

        String greeting = helloMockito.greet("World");

        assertEquals("Hello, World, from Mockito!", greeting);

        }

    }
```

Dado que la clase HelloMockito no tiene dependencias, Mockito no es necesario en absoluto. Para ver dónde es útil Mockito, agreguemos un par de dependencias, una para buscar una persona a quien podamos saludar por su nombre y otra para traducir el mensaje resultante a diferentes idiomas:

HelloMockitoRevised.java

```
public class HelloMockito {

    private String greeting = "Hello, %s, from Mockito!";

    // Dependencies
```

```
    private final PersonRepository personRepository;
    private final TranslationService translationService;
    // Constructor to inject the dependencies
    public HelloMockito(PersonRepository personRepository,
                        TranslationService translationService) {
        this.personRepository = personRepository;
        this.translationService = translationService;
    }
    // Method we want to test
    public String greet(int id, String sourceLang, String targetLang) {
    Optional<Person> person = personRepository.findById(id);
    String name = person.map(Person::getFirst).orElse("World");
    return translationService.translate(
        String.format(greeting, name), sourceLang, targetLang);
    }
}
```

Ahora estamos llegando a alguna parte. El argumento del método greet ahora pretende ser un ID entero para un objeto Person (un objeto Java simple y antiguo, o POJO, que incluye un nombre, un apellido y una fecha de nacimiento). Recuperamos Person de una clase de repositorio que representa algún tipo de almacenamiento persistente. Además, una vez que formamos el saludo de bienvenida que se devolverá, pasamos esa cadena a través de un servicio de traducción y enviamos el resultado.

Aquí están las dos dependencias como interfaces Java. Primero, PersonRepository, que tiene muchos más métodos además del método findById necesario en HelloMockito:

PersonRepository.java

```
    public interface PersonRepository {
    Person save(Person person);
    Optional<Person> findById(int id);
    List<Person> findAll();
    long count();
```

```
    void delete(Person person);
}
```

Como estás familiarizado con el marco de Spring Data, estos métodos te resultarán familiares. Usaremos esta interfaz, combinada con PersonService, en varios ejemplos siguientes. También veremos cómo usar Mockito con Spring.)

El método que necesita nuestra implementación de HelloMockito es findById, que toma una identificación entera y devuelve un Optional envuelto alrededor de Person si la identificación corresponde a una fila existente en la base de datos, o un Optional vacío en caso contrario. El método de bienvenida en HelloMockito extrae el nombre de la persona si la persona existe o usa de forma predeterminada la palabra World si no.

El siguiente es TranslationService, que también es una interfaz, porque hay muchos servicios disponibles públicamente para elegir, incluido el nuestro propio:

TranslationService.java

```
public interface TranslationService{
  default String translate(String text,
                String sourceLang;
                String targetLang); {
        return text;
        }
}
```

Nuestra implementación predeterminada es devolver la cadena de entrada. Si tuviéramos que implementar esto en un sistema real, sería mucho más complicado.

La clase HelloMockito ahora es mucho más difícil de probar porque tenemos que hacer algo con esas dos dependencias. Consigamos ayuda de Mockito. Mockito generará implementaciones de nuestras dependencias para nosotros, que podemos configurar para hacer lo que queramos. Luego podemos inyectar esas dependencias en nuestra clase bajo prueba y actualizar nuestras pruebas en consecuencia.

Agregar Mockito al proyecto

Primero, para que Mockito esté disponible en nuestro proyecto, actualizaremos el archivo de compilación para incluir Mockito. El siguiente ejemplo usa Gradle como herramienta de compilación, pero la versión Maven usa las mismas coordenadas para las dependencias JUnit 5 y Mockito.

Puedes utilizar el archivo libs.versions.toml

```
[versions]
assertj = "3.25.3"
gson = "2.10.1"
jackson = "2.17.0"
junit = "5.11.0-M1"
junit-platform = "1.11.0-M1"
mockito = "5.11.0"
retrofit = "2.11.0"

[libraries]
assertj = { module = "org.assertj:assertj-core", version.ref = "assertj" }
gson = { module = "com.google.code.gson:gson", version.ref = "gson" }
jackson = { module = "com.fasterxml.jackson.core:jackson-databind", version.ref = "jackson" }
junit-jupiter = { module = "org.junit.jupiter:junit-jupiter", version.ref = "junit" }
junit-platform = { module = "org.junit.platform:junit-platform-launcher", version.ref = "junit-platform" }
junit-vintage = { module = "org.junit.vintage:junit-vintage-engine", version.ref = "junit" }
mockito-core = { module = "org.mockito:mockito-core", version.ref = "mockito" }
mockito-junit = { module = "org.mockito:mockito-junit-jupiter", version.ref = "mockito" }
retrofit-core = { module = "com.squareup.retrofit2:retrofit", version.ref = "retrofit" }
retrofit-gson = { module = "com.squareup.retrofit2:converter-gson", version.ref = "retrofit" }

[bundles]
junit = [
    "junit-jupiter",
    "junit-platform",
    "junit-vintage",
]
```

```
mockito = [
   "mockito-core",
   "mockito-junit",
]
```

```
[plugins]
version-catalog-update = "nl.littlerobots.version-catalog-update:0.8.4"
versions = "com.github.ben-manes.versions:0.51.0"
```

Ejercicio: Utiliza este archivo y configura el settings.gradle de la siguiente manera

```
dependencyResolutionManagement {
   versionCatalogs {
      libs {
         from(files("gradle/libs.versions.toml"))
      }
   }
}
```

Luego construye el archivo build.gradle, para colocar todas las dependencias usando el archivo toml. Comprueba que funciona igual que un único archivo build.gradle.

El archivo TOML contiene los números de versión para cada dependencia y crea la referencia de bibliotecas utilizada en el archivo de compilación build.gradle.

Como usamos JUnit 5 para las pruebas, incluimos dos dependencias para Mockito:

- La dependencia de Mockito-Core, y
- La extensión JUnit 5 para Mockito, aquí llamada mockito-junit-jupiter.

Finalmente, necesitamos la llamada al método useJUnitPlatform para decirle a Gradle que estamos usando JUnit 5 para nuestras pruebas; de lo contrario, las pruebas no se detectarán durante la compilación de Gradle. ¡Ahora estamos listos para reunir todo para una prueba completa de Hello, World!

Una clase de prueba de Mockito con todo

Anotemos la clase de prueba resultante con @ExtendWith(MockitoExtension.class) y actualicemos las pruebas para establecer las expectativas en cada mock.

La siguiente prueba lo tiene todo: la extensión Mockito JUnit 5, las anotaciones @Mock y @InjectMocks, establecer expectativas en los stubs generados usando la sintaxis when/thenReturn e incluso verificar que los métodos mocks fueron llamados la cantidad correcta de veces en el orden correcto.

Sin más, el ejemplo completo ¡Hello World! :

HelloMockitoTestFull.java

```
@ExtendWith(MockitoExtension.class)
class HelloMockitoTest {
	@Mock
	private PersonRepository repository;
	@Mock
	private TranslationService translationService;
	@InjectMocks
	private HelloMockito helloMockito;
@Test
@DisplayName("Greet Admiral Hopper")
void greetAPersonThatExists() {
	// set the expectations on the mocks
	when(repository.findById(anyInt()))
		.thenReturn(Optional.of(new Person(1, "Grace", "Hopper",
			LocalDate.of(1906, Month.DECEMBER, 9))));
	when(translationService
		.translate("Hello, Grace, from Mockito!", "en", "en"))
		.thenReturn("Hello, Grace, from Mockito!");
	// test the greet method
	String greeting = helloMockito.greet(1, "en", "en");
	assertEquals("Hello, Grace, from Mockito!", greeting);
	// verify the methods are called once, in the right order
	InOrder inOrder = inOrder(repository, translationService);
```

```
        inOrder.verify(repository).findById(anyInt());
        inOrder.verify(translationService)
                .translate(anyString(), eq("en"), eq("en"));
        }
        @Test
        @DisplayName("Greet a person not in the database")
        void greetAPersonThatDoesNotExist() {
                when(repository.findById(anyInt()))
                        .thenReturn(Optional.empty());
                when(translationService
                        .translate("Hello, World, from Mockito!", "en", "en"))
                        .thenReturn("Hello, World, from Mockito!");
                String greeting = helloMockito.greet(100, "en", "en");
                assertEquals("Hello, World, from Mockito!", greeting);
        // verify the methods are called once, in the right order
        InOrder inOrder = inOrder(repository, translationService);
        inOrder.verify(repository).findById(anyInt());
        inOrder.verify(translationService)
                .translate(anyString(), eq("en"), eq("en"));
        }
}
```

Una de las ventajas de usar una biblioteca como Mockito es que podemos probar una clase con dependencias incluso antes de implementarlas. Este es el patrón que usaremos cuando una clase tenga dependencias:

```
class ClassUnderTest {
    private DependencyOne dependency1;
    private DependencyTwo dependency2;
     public Result methodToBeTested(Object... args) {
```

```
    // use dependencies and arguments and return Result
     return result;
      }
   }
```

Podemos probar el methodToBeTested con los mocks de las dos dependencias, inyectándolas en la clase bajo prueba y luego llamando al método de prueba.

Proceso de prueba de Mockito

Formalmente, el proceso que estamos utilizando consiste en:

- Crear stubs para sustituir las dependencias.
- Establecer expectativas en los talones para hacer lo que quieras.
- Inyectar los stubs en la clase que planeas probar.
- Probar los métodos de la clase bajo prueba invocando sus métodos, que a su vez llaman a métodos en los stubs.
- Verificar que los métodos funcionen como se esperaba.
- Verificar que los métodos de las dependencias se hayan invocado la cantidad correcta de veces y en el orden correcto.

Puede seguir estos pasos cada vez que desee sutilizar Mockito (o, honestamente, cualquier herramienta similar) para reemplazar dependencias en la clase bajo prueba, escribiendo así pruebas unitarias verdaderas.

Ejercicio 1: Simulación de excepciones

Crea una prueba que simule una excepción lanzada por el PersonRepository y asegúrate de que tu código maneje la excepción correctamente.

- Modifica HelloMockito para que maneje excepciones lanzadas por PersonRepository.
- Crea un nuevo método de prueba greetPersonThrowsException en HelloMockitoTestFull.java para simular una excepción y verificar el comportamiento del método greet.

Ejercicio 2: Verificación de llamadas a métodos

Asegúrate de que el método translate no se llame si PersonRepository lanza una excepción.

- Modifica HelloMockito para que no llame a translationService.translate si ocurre una excepción.
- Crea una prueba para verificar este comportamiento.

Ejercicio 3: Prueba de traducción en diferentes idiomas

Asegúrate de que el servicio de traducción funciona correctamente para diferentes combinaciones de idiomas.

- Modifica HelloMockito para incluir más lógica de traducción.
- Crea pruebas que verifiquen la traducción a diferentes idiomas.

Ejercicio 4: Uso de AssertJ para verificaciones más complejas

Utiliza AssertJ para realizar verificaciones más complejas en tus pruebas, como verificaciones en listas o en objetos anidados.

- Añade un método en HelloMockito que retorne una lista de saludos para múltiples personas.
- Crea pruebas que verifiquen la lista de saludos utilizando AssertJ.

Parte 2

Contando astronautas por nave espacial

¿Cuántos astronautas hay en cada nave en el espacio en este momento? Hablemos de una solución Java moderna que no requiere muchas clases pero utiliza una arquitectura familiar que encontrará en muchas aplicaciones. Usaremos este problema como una introducción a las pruebas en general y a Mockito en particular.

El producto final tendrá tres componentes:

- Una clase llamada AstroGateway que accede a un servicio web RESTful para recuperar los datos del astronauta en formato JSON y luego convierte la estructura JSON en Java POJO (objetos Java simples y antiguos).
- Una clase llamada AstroService que procesa los POJO de Java y devuelve un Map de cadenas a números enteros, donde las claves del mapa son los nombres de las naves espaciales y los valores del mapa son el número de astronautas a bordo de cada una.
- Una clase de aplicación que impulsa el proceso, que en nuestro caso será una serie de casos de prueba que invocan los métodos correctos e imprimen y verifican los resultados.

Una nota sobre las versiones de Java

Mockito solo requiere Java 8. La versión actual de soporte a largo plazo (LTS) de Java es 17, pero a partir de 2022, la comunidad se divide entre Java 8 y la versión LTS anterior, Java 11.

La implementación de este sistema “astro” utiliza el cliente HTTP agregado a Java en la versión 11, y 11 es la versión predeterminada de Java que usaremos a lo largo de la actividad. Ninguno de los códigos

utiliza registros, clases selladas, coincidencia de patrones, expresiones de cambio ni ninguna de las otras características más nuevas agregadas desde 11.

Como recordatorio, para agregar Mockito a un proyecto, las únicas dependencias necesarias son mockito-core y, cuando se usa la extensión Mockito JUnit 5, como aquí, mockito-junit-jupiter.

Con el proyecto configurado para pruebas, agreguemos las clases para hacer el trabajo.

Creando las clases básicas

Los datos de los astronautas provienen del servicio web RESTful gratuito de Open Notify llamado People in Space. Solo admite solicitudes HTTP GET, pero es gratuito y no requiere registro ni ningún tipo de clave. Devuelve una respuesta JSON que muestra todos los astronautas en el espacio en un momento dado. Si envía una solicitud HTTP GET a http://api.open-notify.org/astros.json, recibirá una respuesta similar a esta:

astro_data.json


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

```
    "number": 10
}
```

Como puedes ver en la muestra, la respuesta incluye el número total de astronautas y una colección llamada people que contiene combinaciones de nombres y naves para cada astronauta. Los POJO de Java crean una estructura conveniente para contener la información JSON analizada. Sólo se necesitan dos clases de Java:

- Assignment, que representa el par name y craft del astronauta
- AstroResponse, que muestra el número total de personas en el espacio, un mensaje de éxito y la lista de personas o lista de asignaciones.

Las clases de Java se implementarán como POJO tradicionales, lo que significa que tendrán atributos que coincidan con las propiedades JSON, junto con métodos getter y setter. Aquí está la clase de

Assignment.java:

```
public class Assignment {
    private final String name;
    private final String craft;
    // constructors, getters and setters, toString
}
```

Aquí está la clase AstroResponse.java:

```
public class AstroResponse {
  private final int number;
    private final String message;
    private final List<Assignment> people;
// constructors, getters and setters, toString
}
```

Para acceder al servicio web RESTful, utilizaremos el patrón de diseño Gateway. Un gateway es una clase que encapsula el acceso a un recurso externo. En este caso particular, solo necesitamos una única puerta de enlace, llamada AstroGateway, que se conecta al servicio remoto, descarga los datos astro y convierte la respuesta JSON en registros. Java es más feliz cuando usa interfaces, y hacerlo aquí facilitará las pruebas porque podremos sustituir un gateway falso cuando la necesitemos. Por lo tanto, agrea una interfaz gateway que contenga un único método llamado getResponse:

Gateway.java

```
public interface Gateway<T> {

        T getResponse();

}
```

El método getResponse devuelve una instancia de tipo genérico.

La clase concreta que implementará la interfaz Gateway será AstroGateway. Lo implementará utilizando la API HttpClient si está utilizando Java 11+ o la biblioteca Retrofit 2[8] si todavía está en Java 8.

Aquí estamos definiendo las clases básicas, así que definamos también la clase AstroService, cuyo trabajo es convertir los registros devueltos por el gateway en un mapa Java. La clase necesita solo un método, llamado getAstroData, con la siguiente firma:

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

La clase AstroService utiliza el gateway para acceder al servicio web remoto. El gateway recupera los datos remotos y los convierte en una instancia Success que contiene una AstroResponse o en una instancia Failure que contiene cualquier excepción lanzada, en caso contrario. Suponiendo que tiene éxito, el servicio extrae los datos que queremos y los convierte en un mapa, donde las claves son los nombres de las naves espaciales y los valores son el número de astronautas a bordo de cada una.

Al final, nuestra aplicación crea una instancia de AstroService, llama a su método getAstroData y procesa los resultados. Los resultados (basados en los datos JSON mostrados anteriormente) serán los siguientes:

```
3 astronautas a bordo del Tiangong

7 astronautas a bordo de la ISS
```

Ahora pasemos a las pruebas necesarias para verificar que la implementación funcione como se desea.

Agregar pruebas unitarias y de integración (de un extremo a otro)

Para nuestros propósitos, una prueba de integración es aquella que prueba un sistema, incluidos los componentes con los que interactúa (dependencias), mientras que una prueba unitaria es aquella que prueba una clase de forma aislada, sin interacciones ni dependencias.

Las pruebas de integración suelen ser más fáciles de entender porque se trata el sistema tal como se utiliza en la práctica, sin información adicional sobre su implementación. En este caso, el sistema está controlado por el método getAstroData en AstroService, que utiliza un AstroGateway. Su prueba de integración se ve así:

AstroServiceTest.java

```
@Test

void testAstroData_usingRealGateway_withHttpClient() {
```

```
    // Create an instance of AstroService using the real Gateway
        service = new AstroService(new AstroGateway());
    // Call the method under test
        Map<String, Long> astroData = service.getAstroData();
         // Print the results and check that they are reasonable
                astroData.forEach((craft, number) -> {
                        System.out.println(number + " astronauts aboard " + craft);
        assertAll(
                () -> assertThat(number).isPositive(),
                () -> assertThat(craft).isNotBlank()
              );
        });
}
```

Agregamos AstroGateway a AstroService usando un argumento de constructor. Si queremos reemplazar el AstroGateway real con una instancia falsa, podemos usar ese constructor para inyectarlo en nuestro servicio. El punto importante es que el gateway es una dependencia del servicio.

Para usar Mockito, primero debes determinar las dependencias de la clase que estás probando. Para AstroService, la única dependencia es Gateway.

La prueba utiliza el servicio para llamar al método getAstroData y verifica el mapa devuelto. La prueba utiliza el método forEach(BiConsumer) agregado a la interfaz Map en Java 8, que separa automáticamente las claves de los valores, por lo que es bastante fácil imprimir y verificar los resultados. Luego, la prueba confirma los valores devueltos utilizando aseveraciones de la biblioteca AssertJ.[9] El resultado verifica la combinación de las clases AstroService y AstroGateway, lo cual es necesario si queremos creer que el sistema está funcionando correctamente.

En nuestra prueba de integración, adoptamos un enfoque práctico. El propósito de las pruebas es aislar los problemas. Si tuviéramos una prueba de integración fallida, usaríamos pruebas unitarias para aislar la falla en el gateway o en el servicio.

Las pruebas de integración fallidas conducen a pruebas unitarias

La pregunta clave es siempre: "¿Cómo sabes que tienes razón?" Si una prueba falla, sabes que hay un problema, pero ¿qué tan fácil es localizarlo? Es necesario realizar pruebas de integración exitosas; de lo contrario, no sabrá que el sistema funciona.

Las pruebas de integración fallidas son las que le llevan a las pruebas unitarias, porque si una prueba unitaria falla, sabrá rápidamente dónde y por qué.

Digamos que nuestra prueba de integración falló y ahora queremos crear una prueba para AstroService.

Un stub para el gateway

Una verdadera prueba unitaria para AstroService necesita un sustituto para el gateway: algún tipo de objeto falso que devolverá exactamente lo que queremos cuando el servicio llame a su método getResponse. No queremos que ningún problema con el gateway afecte el éxito o el fracaso de nuestras pruebas del servicio. Este sustituto de el gateway se llama objeto falso (fake). Desde el punto de vista de Mockito, el objeto falso es un mock o un stub dependiendo de cómo se use.

Antes de llegar a esa distinción, siempre podemos proporcionar nuestro propio objeto falso para la dependencia. En este caso, aquí hay una clase llamada FakeGateway que codifica una respuesta:

FakeGateway.java

```
public class FakeGateway implements Gateway<AstroResponse> {
        @Override
        public AstroResponse getResponse() {
                return new AstroResponse(7, "Success",
                  List.of(new Assignment("Kathryn Janeway", "USS Voyager"),
                          new Assignment("Seven of Nine", "USS Voyager"),
                          new Assignment("Will Robinson", "Jupiter 2"),
                          new Assignment("Lennier", "Babylon 5"),
                          new Assignment("James Holden", "Rocinante"),
                          new Assignment("Naomi Negata", "Rocinante"),
        new Assignment("Ellen Ripley", "Nostromo")));
        }
}
```

Aquí hay una prueba unitaria del servicio que utiliza FakeGateway:

AstroServiceTest.java

```
@Test
```

```
  void testAstroData_usingOwnMockGateway() {
   // Create the service using the mock Gateway
       service = new AstroService(new MockGateway());
   // Call the method under test
    Map<String, Long> astroData = service.getAstroData();
   // Check the results from the method under test
          astroData.forEach((craft, number) -> {
                System.out.println(number + " astronauts aboard " + craft);
                   assertAll(
                            () -> assertThat(number).isPositive(),
                             () -> assertThat(craft).isNotBlank()
              );
            });
         }
```

Dada la respuesta devuelta por el mock gateway, el servicio ahora imprime:

1 astronautas a bordo de Babylon 5

1 astronautas a bordo del Nostromo

1 astronautas a bordo de Júpiter 2

2 astronautas a bordo del USS Voyager

2 astronautas a bordo de Rocinante

Si bien en este caso fue bastante fácil escribir nuestra propio mock gateway , vale la pena mencionar dos puntos:

- Un trabajo de Mockito es automatizar la creación de objetos falsos como nuestro FakeGateway. Mockito generará una clase que implementa la interfaz Gateway por nosotros. Luego podemos decirle qué devolver cuando se llama al método getResponse, y el resto sucede automáticamente.
- Nuestro FakeGateway es un mock o un stub, dependiendo de cómo se utilice.

Ahora que hemos visto una prueba de integración y una prueba con nuestro propio objeto stubbed, estamos listos para usar Mockito para automatizar el proceso. Después de abordar los conceptos

básicos de la API de Mockito, volveremos al proyecto Astro para ver cómo escribir pruebas unitarias reales para el servicio.

Hasta ahora, hemos introducido que Mockito genera clases falsas que sustituyen a dependencias reales para que puedas aislar la clase que estás probando. Hemos hablado sobre el uso de stubs cuando necesitamos simplemente proporcionar salidas conocidas para entradas conocidas. Cuando queremos comprobar que la clase bajo prueba llamó a los métodos del mock la cantidad correcta de veces en el orden correcto, sabemos que debemos usar un mock. En resumen, hemos explicado por qué usarías el marco Mockito.

Ahora estamos listos para iniciar el proceso de uso de Mockito en las pruebas. En las evaluacioens, crearemos mocks, los inyectaremos en la clase bajo prueba y haremos que respondan de la manera que queremos.

Ejercicio 1: Prueba de integración con datos reales

Crea una prueba de integración para AstroService que use un RealGateway que haga una llamada HTTP real a una API externa para obtener datos sobre astronautas. Verifica que los datos obtenidos sean correctos.

1. Implementa RealGateway que haga una llamada HTTP a una API real.
2. Crea una prueba de integración para AstroService que utilice RealGateway.
3. Verifica que la respuesta contenga datos válidos.

```
public class RealGateway implements Gateway<AstroResponse> {
   @Override
   public AstroResponse getResponse() {
      // Realiza una llamada HTTP a una API real y convierte la respuesta en AstroResponse
   }
}
```

```
@Test
@DisplayName("Integration test with real API")
void testAstroData_usingRealGateway_withRealData() {
   service = new AstroService(new RealGateway());
```

```
    Map<String, Long> astroData = service.getAstroData();
    astroData.forEach((craft, number) -> {
        System.out.println(number + " astronauts aboard " + craft);
        assertAll(
            () -> assertThat(number).isPositive(),
            () -> assertThat(craft).isNotBlank()
        );
    });
}
```

Ejercicio 2: Prueba con datos vacíos

Crea una prueba para AstroService que utilice un FakeGateway que devuelva una lista vacía de astronautas. Verifica que el servicio maneje correctamente este caso y no falle.

- Modifica FakeGateway para devolver una respuesta con una lista vacía.
- Crea una prueba que utilice esta versión de FakeGateway.
- Verifica que AstroService maneje correctamente la lista vacía.

```
public class EmptyFakeGateway implements Gateway<AstroResponse> {
    @Override
    public AstroResponse getResponse() {
        return new AstroResponse(0, "No astronauts", List.of());
    }
}

@Test
@DisplayName("Handle empty astronaut list")
void testAstroData_withEmptyGateway() {
    service = new AstroService(new EmptyFakeGateway());
    Map<String, Long> astroData = service.getAstroData();
    assertTrue(astroData.isEmpty(), "Astro data should be empty");
}
```

Ejercicio 3: Prueba de fallo en el gateway

Crea una prueba para AstroService que utilice un FakeGateway que lance una excepción cuando se llama a getResponse(). Verifica que AstroService maneje esta excepción correctamente.

- Modifica FakeGateway para lanzar una excepción en getResponse().
- Crea una prueba que utilice esta versión de FakeGateway.
- Verifica que AstroService maneje correctamente la excepción.

```
public class ExceptionFakeGateway implements Gateway<AstroResponse> {
   @Override
   public AstroResponse getResponse() {
      throw new RuntimeException("Gateway failure");
   }
}
```

```
@Test
@DisplayName("Handle gateway exception")
void testAstroData_withExceptionGateway() {
   service = new AstroService(new ExceptionFakeGateway());
   Map<String, Long> astroData = service.getAstroData();
   assertTrue(astroData.isEmpty(), "Astro data should be empty due to exception");
}
```

Ejercicio 4: Verificación de datos duplicados

Crea una prueba para AstroService que verifique que no hay datos duplicados en la lista de astronautas proporcionada por el gateway.

- Modifica FakeGateway para incluir datos duplicados.
- Crea una prueba que utilice esta versión de FakeGateway.
- Verifica que AstroService maneje correctamente los datos duplicados.

Ejercicio 5: Prueba de filtrado por nave espacial

Crea un método en AstroService que filtre los astronautas por nave espacial y verifica su funcionamiento mediante pruebas unitarias.

- Añade un método en AstroService que filtre astronautas por nave espacial.
- Crea pruebas para este método utilizando FakeGateway.