EBook Gratis

APRENDIZAJE mockito

Free unaffiliated eBook created from
Stack Overflow contributors.

#mockito

Tabla de contenido

Acerca de

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: mockito

It is an unofficial and free mockito ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official mockito.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Capítulo 1: Empezando con mockito

Observaciones

Mockito es un marco de simulación de Java que tiene como objetivo proporcionar la capacidad de escribir con claridad una prueba de unidad legible utilizando su API simple. Se diferencia de otros marcos de simulacros al dejar el patrón de esperar-ejecutar-verificar que la mayoría de los otros marcos utilizan.

En su lugar, solo conoce una forma de simular las clases e interfaces (no final) y permite verificar y apilar basándose en comparadores de argumentos flexibles.

La versión actual 1.10.19 se obtiene mejor utilizando maven

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>1.10.19</version>
</dependency>
```

o gradle

```
repositories { jcenter() }
dependencies { testCompile "org.mockito:mockito-core:1.+" }
```

La versión 2 todavía está en beta.

Versiones

Versión	Maven Central	Notas de lanzamiento	Fecha de lanzamiento
2.1.0	mockito-core	cambios	2016-10-04
1.10.19	mockito-core	cambios	2014-12-31

Examples

Prueba unitaria simple utilizando Mockito.

La clase que vamos a probar es:

```
public class Service {

    private Collaborator collaborator;

    public Service(Collaborator collaborator) {
```

```
        this.collaborator = collaborator;
    }

    public String performService(String input) {
        return collaborator.transformString(input);
    }
}
```

Su colaborador es:

```
public class Collaborator {

    public String transformString(String input) {
        return doStuff();
    }

    private String doStuff() {
        // This method may be full of bugs
        . . .
        return someString;
    }

}
```

En nuestra prueba, queremos romper la dependencia de `Collaborator` y sus errores, por lo que vamos a burlarnos de `Collaborator` :

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

import org.junit.Test;

public class ServiceTest {
    @Test
    public void testPerformService() throws Exception {
        // Configure mock
        Collaborator collaboratorMock = mock(Collaborator.class);
        doReturn("output").when(collaboratorMock).transformString("input");

        // Perform the test
        Service service = new Service(collaboratorMock);
        String actual = service.performService("input");

        // Junit asserts
        String expected = "output";
        assertEquals(expected, actual);
    }
}
```

Usando las anotaciones de Mockito

La clase que vamos a probar es:

```
public class Service{

    private Collaborator collaborator;
```

```
    public Service(Collaborator collaborator){
        this.collaborator = collaborator;
    }


    public String performService(String input){
        return collaborator.transformString(input);
    }
}
```

Su colaborador es:

```
public class Collaborator {

    public String transformString(String input){
        return doStuff();
    }

    private String doStuff()
    {
        // This method may be full of bugs
        . . .
        return someString;
    }

}
```

En nuestra prueba, queremos romper la dependencia de `Collaborator` y sus errores, por lo que vamos a burlarnos de `Collaborator` . Usar la anotación de `@Mock` es una manera conveniente de crear diferentes instancias de simulacros para cada prueba:

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

import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.InjectMocks;
import org.mockito.runners.MockitoJUnitRunner;

@RunWith(MockitoJUnitRunner.class)
public class ServiceTest {

    @Mock
    private Collaborator collaboratorMock;

    @InjectMocks
    private Service service;

    @Test
    public void testPerformService() throws Exception {
        // Configure mock
        doReturn("output").when(collaboratorMock).transformString("input");

        // Perform the test
        String actual = service.performService("input");
```

```
        // Junit asserts
        String expected = "output";
        assertEquals(expected, actual);
    }


    @Test(expected=Exception.class)
    public void testPerformServiceShouldFail() throws Exception {
        // Configure mock
        doThrow(new Exception()).when(collaboratorMock).transformString("input");

        // Perform the test
        service.performService("input");
    }
}
```

Mockito intentará resolver la inyección de dependencia en el siguiente orden:

1. **Inyección basada** en el constructor: los simulacros se inyectan en el constructor con la mayoría de los argumentos (si no se pueden encontrar algunos argumentos, se pasan nulos). Si un objeto fue creado exitosamente a través del constructor, entonces no se aplicarán otras estrategias.
2. **Inyección a base de Setter** - los mock son inyectados por tipo. Si hay varias propiedades del mismo tipo, los nombres de las propiedades y los nombres simulados coincidirán.
3. **Inyección directa en el campo** - igual que para la inyección basada en un fijador

Tenga en cuenta que no se informa de ninguna falla en caso de que alguna de las estrategias mencionadas fallara.

Consulte la última `@InjectMocks` de `@InjectMocks` para obtener información más detallada sobre este mecanismo en la última versión de Mockito.

Instalación y configuración

Instalación

La forma preferida de instalar Mockito es declarar una dependencia de `mockito-core` con un sistema de compilación de elección. A partir del 22 de julio de 2016, la última versión no beta es 1.10.19, pero se recomienda migrar a la versión 2.x.

Maven

```
<dependency>
    <groupId>org.mockito</groupId>
    <artifactId>mockito-core</artifactId>
    <version>1.10.19</version>
    <scope>test</scope>
</dependency>
```

Gradle

```
repositories { jcenter() }
dependencies { testCompile "org.mockito:mockito-core:1.+" }
```

También hay `mockito-all` que contiene Hamcrest y Objenesis además de Mockito. Se entrega a través de Maven principalmente para usuarios de hormigas, pero la distribución se ha suspendido en Mockito 2.x.

Importar

La mayoría de las instalaciones de Mockito son métodos estáticos de `org.mockito.Mockito` . Por lo tanto, Mockito se puede importar estáticamente en una clase de esta manera:

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

El punto de entrada de la documentación se encuentra en el javadoc de esta clase.

Se burlan de algunos métodos en un objeto

Solo algunos métodos de un objeto se pueden burlar usando el `spy()` de mockito.

Por ejemplo, imagine que la clase de método requiere algún servicio web para funcionar.

```
public class UserManager {

    List<User> users;

    public UserManager() {
        user = new LinkedLisk<User>();
    }

    public void addUser(User user) {
        if (isValid(user)) {
            user.add(user);
        } else {
            throw new NotValidUserException();
        }
    }

    protected boolean isValid(User user) {
        //some online web service to check if user is valid
    }

    public int numberOfUsers() {
        return users.size();
    }
}
```

`addUser` método `addUser` se debe probar para realizar una prueba útil para `UserManager` . Sin embargo, aquí se encuentra una dependencia, `isValid` requiere un servicio web externo que no está contenido en nuestro código. Entonces, esta dependencia externa debe ser neutralizada.

En este caso, si solo se burla de `isValid` , podrá probar el resto de los métodos de `UserManager` .

```
@Test
public void testAddUser() {
    User user = mock(User.class);
    UserManager  manager = spy(new UserManager());

    //it forces to manager.isValid to return true
    doReturn(true).when(manager).isValid(anyObject());

    manager.addUser(user);
    assertTrue(manager.numberOfUsers(), 1);
}
```

Puede comprobar fácilmente el escenario donde el `user` no es válido.

```
@Test(expectedExceptions = NotValidUserException.class)
public void testNotValidAddUser() {
    User user = mock(User.class);
    UserManager  manager = spy(new UserManager());

    //it forces to manager.isValid to return false
    doReturn(false).when(manager).isValid(anyObject());

    manager.addUser(user);
}
```

Prueba de Mockito mínimo simple

Este ejemplo muestra una prueba de Mockito mínima usando una `ArrayList` simulada:

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

import java.util.ArrayList;

import org.junit.Test;
import org.junit.runner.RunWith;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;

@RunWith(MockitoJUnitRunner.class)
public class MockitoTest
{
    @Mock
    ArrayList<String> listMock;

    @Test
    public void testAppend() {
        // configure the mock to return "foobar" whenever "get()"
        // is called on "listMock" with an int value as parameter
        doReturn("foobar").when(listMock).get(anyInt());
        String result = listMock.get(0);

        assertEquals("foobar", result);
    }
}
```

Verificando argumentos con ArgumentCaptor

Para validar argumentos a métodos llamados en un simulacro, use la clase `ArgumentCaptor` . Esto le permitirá extraer los argumentos en su método de prueba y realizar aseveraciones en ellos.

Este ejemplo prueba un método que actualiza el nombre de un usuario con un ID determinado. El método carga al usuario, actualiza el atributo de `name` con el valor dado y lo guarda después. La prueba desea verificar que el argumento pasado al método de `save` es un objeto `User` con la ID y el nombre correctos.

```
// This is mocked in the test
interface UserDao {
    void save(User user);
}

@RunWith(MockitoJUnitRunner.class)
public class UserServiceTest {
    @Mock
    UserDao userDao;

    @Test
    public void testSetNameForUser() {
        UserService serviceUnderTest = new UserService(userDao);

        serviceUnderTest.setNameForUser(1L, "John");

        ArgumentCaptor<User> userArgumentCaptor = ArgumentCaptor.forClass(User.class);

        verify(userDao).save(userArgumentCaptor.capture());
        User savedUser = userArgumentCaptor.getValue();
        assertTrue(savedUser.getId() == 1);
        assertTrue(savedUser.getName().equals("John"));
    }
}
```

Verificación de argumentos con ArgumentMatcher

Mockito proporciona una interfaz `Matcher<T>` junto con una clase abstracta `ArgumentMatcher<T>` para verificar los argumentos. Utiliza un enfoque diferente para el mismo caso de uso que el `ArgumentCaptor` . Además, el ArgumentMatcher también se puede utilizar para burlarse. Ambos casos de uso hacen uso del método `Mockito.argThat()` que proporciona un código de prueba razonablemente legible.

```
verify(someMock).someMethod(Mockito.argThat(new ArgumentMatcher<String>() {

    @Override
    public boolean matches(Object o) {
        return o instanceof String && !((String)o).isEmpty();
    }

});
```

Desde los JavaDocs de ArgumentMatcher:

Advertencia: Sea razonable con el uso de la coincidencia de argumentos complicados,

especialmente los emparejadores de argumentos personalizados, ya que puede hacer que la prueba sea menos legible. A veces es mejor implementar equals () para los argumentos que se pasan a simulacros (Mockito, naturalmente, usa equals () para la coincidencia de argumentos). Esto puede hacer que la prueba sea más limpia.

Crea objetos burlados por mockito.

Hay dos formas de crear un objeto burlado por Mockito:

- a través de la anotación
- a través de la función simulada

A través de la anotación:

Con un corredor de prueba JUnit:

```
@RunWith(MockitoJUnitRunner.class)
public class FooTest {
    @Mock
    private Bar barMock;

    // ...
}
```

También puede usar la función JUnit `@Rule` , que proporciona la misma funcionalidad que `MockitoJUnitRunner` , pero no necesita un `@RunWith` prueba `@RunWith` :

```
public class FooTest {
    @Rule
    public MockitoRule mockito = MockitoJUnit.rule();

    @Mock
    private Bar barMock;

    // ...
}
```

Si no puede usar `@RunWith` o la anotación `@Rule` , también puede `@Rule` "por mano":

```
public class FooTest {
    @Mock
    private Bar barMock;

    @Before
    public void setUp() {
        MockitoAnnotations.initMocks(this);
    }

    // ...
}
```

A través de la función simulada:

```
public class FooTest {
    private Bar barMock = Mockito.mock(Bar.class);

    // ...
}
```

Debido al borrado de tipo, no puede simular una clase genérica como se indicó anteriormente. Debe burlarse de la clase base y convertir explícitamente al tipo genérico correcto:

```
public class FooTest {
    private Bar<String> genericBarMock = (Bar<String>) Mockito.mock(Bar.class);

    // ...
}
```

Añadir comportamiento a objeto simulado

```
Mockito.when(mock.returnSomething()).thenReturn("my val");

mock.returnSomething(); // returns "my val"
mock.returnSomething(); // returns "my val" again
mock.returnSomething(); // returns "my val" again and again and again...
```

Si desea un valor diferente en la segunda llamada, puede agregar el argumento de retorno deseado al método de Retorno:

```
Mockito.when(mock.returnSomething()).thenReturn("my val", "other val");

mock.returnSomething(); // returns "my val"
mock.returnSomething(); // returns "other val"
mock.returnSomething(); // returns "other val" again
```

Si llama al método sin agregar comportamiento al simulacro, devolverá nulo:

```
barMock.mock.returnSomethingElse(); // returns null
```

En caso de que el método simulado tenga parámetros, también debe declarar los valores:

```
Mockito.when(mock.returnSomething("param 1")).thenReturn("my val 1");
Mockito.when(mock.returnSomething("param 2")).thenReturn("my val 2");

mock.returnSomething("param 1"); // returns "my val 1"
mock.returnSomething("param 2"); // returns "my val 2"
mock.returnSomething("param 3"); // returns null
```

Si no te importa el valor param, puedes usar Matchers.any ():

```
Mockito.when(mock.returnSomething(Matchers.any())).thenReturn("p1");

mock.returnSomething("param 1"); // returns "p1"
mock.returnSomething("param other"); // returns "p1"
```

Para lanzar la excepción use el método thenThrow:

```
Mockito.when(mock.returnSomething()).thenThrow(new Exception());

mock.returnSomething(); // throws Exception
```

Revise los argumentos pasados a simular

Asumamos que tenemos esta clase y nos gustaría probar el método `doSmth` . En este caso, queremos ver si el parámetro "val" se pasa a `foo` . Objeto `foo` se burla.

```
public class Bar {

    private final Foo foo;

    public Bar(final Foo foo) {
        this.foo = foo;
    }

    public void doSmth() {
        foo.bla("val");
    }
}
```

Podemos lograr esto con `ArgumentCaptor` :

```
@Mock
private Foo fooMock;

@InjectMocks
private Bar underTest;

@Captor
private ArgumentCaptor<String> stringCaptor;

@Test
public void should_test_smth() {
    underTest.doSmth();

    Mockito.verify(fooMock).bla(stringCaptor.capture());

    assertThat(stringCaptor.getValue(), is("val"));
}
```

Verifique las llamadas de método en objeto simulado

Para verificar si se llamó a un método en un objeto `Mockito.verify` puede usar el método `Mockito.verify` :

```
Mockito.verify(someMock).bla();
```

En este ejemplo, afirmamos que el método `bla` fue llamado en el objeto simulado `someMock` .

También puede verificar si un método fue llamado con ciertos parámetros:

```
Mockito.verify(someMock).bla("param 1");
```

Si desea verificar que *no se haya* llamado a un método, puede pasar un parámetro adicional de `VerificationMode` para `verify` :

```
Mockito.verify(someMock, Mockito.times(0)).bla();
```

Esto también funciona si desea comprobar que este método fue llamado más de una vez (en este caso, verificamos que el método `bla` fue llamado 23 veces):

```
Mockito.verify(someMock, Mockito.times(23)).bla();
```

Estos son más ejemplos para el parámetro `VerificationMode` , que proporcionan un mayor control sobre el número de veces que se debe llamar un método:

```
Mockito.verify(someMock, Mockito.never()).bla(); // same as Mockito.times(0)

Mockito.verify(someMock, Mockito.atLeast(3)).bla(); // min 3 calls

Mockito.verify(someMock, Mockito.atLeastOnce()).bla(); // same as Mockito.atLeast(1)

Mockito.verify(someMock, Mockito.atMost(3)).bla(); // max 3 calls
```

Métodos de vacío

`void` métodos `void` pueden eliminarse utilizando la familia de métodos doThrow () , doAnswer () , doNothing () , doCallRealMethod () .

```
Runnable mock = mock(Runnable.class);

doThrow(new UnsupportedOperationException()).when(mock).run();

mock.run(); // throws the UnsupportedOperationException
```

Tenga en cuenta que los métodos de `void` no se pueden apagar `when(..)` hacen que al compilador no le gusten los métodos de `void` como argumento.

Lea Empezando con mockito en línea: https://riptutorial.com/es/mockito/topic/2055/empezando-con-mockito

Capítulo 2: Burlarse de

Examples

Simulacro simple

Mockito ofrece un mehtod de talla única para crear simulacros de clases e interfaces (no finales).

```
Dependency mock = Mockito.mock(Dependency.class);
```

Esto crea una instancia simulada de `Dependency` independientemente de si la `Dependency` es una interfaz o una clase.

Entonces es posible aplazar las llamadas de método a ese simulacro usando la notación Mockito.when (x) .thenReturn (y).

```
Mockito.when(mock.possiblyBuggyMethod()).thenReturn("someString");
```

Así que las llamadas a `Dependency.possiblyBuggyMethod()` simplemente devuelven `"someString"` .

Hay otra notación que se desaconseja en la mayoría de los casos de uso, ya que no es de tipo seguro.

```
Mockito.doReturn("someString").when(mock).possiblyBuggyMethod()
```

Simulacros de impagos

Mientras que un simulacro simple devuelve nulo (o valores predeterminados para primitivas) a cada llamada, es posible cambiar ese comportamiento.

```
Dependency mock = Mockito.mock(Dependency.class, new Answer() {

        @Override
        public Object answer(InvocationOnMock invocationOnMock) throws Throwable {
            return "someString";
        }
    });
```

o usando lambdas:

```
Dependency mock = Mockito.mock(Dependency.class, (Answer) invocationOnMock -> "someString");
```

Este ejemplo devuelve "someString" a cada invocación, pero es posible definir cualquier lógica en el método de respuesta.

Burlándose de una clase usando anotaciones

Clase a prueba:

```
public class GreetingsService { // class to be tested in isolation
    private UserService userService;

    public GreetingsService(UserService userService) {
        this.userService = userService;
    }

    public String getGreetings(int userId, LocalTime time) { // the method under test
        StringBuilder greetings = new StringBuilder();
        String timeOfDay = getTimeOfDay(time.getHour());
        greetings.append("Good ").append(timeOfDay).append(", ");
        greetings.append(userService.getFirstName(userId)) // this call will be mocked
                .append(" ")
                .append(userService.getLastName(userId)) // this call will be mocked
                .append("!");
        return greetings.toString();
    }

    private String getTimeOfDay(int hour) { // private method doesn't need to be unit tested
        if (hour >= 0 && hour < 12)
            return "Morning";
        else if (hour >= 12 && hour < 16)
            return "Afternoon";
        else if (hour >= 16 && hour < 21)
            return "Evening";
        else if (hour >= 21 && hour < 24)
            return "Night";
        else
            return null;
    }
}
```

El comportamiento de esta interfaz será burlado:

```
public interface UserService {
    String getFirstName(int userId);

    String getLastName(int userId);
}
```

Supongamos la implementación real del `UserService` :

```
public class UserServiceImpl implements UserService {
    @Override
    public String getFirstName(int userId) {
        String firstName = "";
        // some logic to get user's first name goes here
        // this could be anything like a call to another service,
        // a database query, or a web service call
        return firstName;
    }

    @Override
    public String getLastName(int userId) {
        String lastName = "";
        // some logic to get user's last name goes here
```

```
        // this could be anything like a call to another service,
        // a database query, or a web service call
        return lastName;
    }
}
```

Prueba de Junit con Mockito:

```
public class GreetingsServiceTest {
    @Mock
    private UserServiceImpl userService; // this class will be mocked
    @InjectMocks
    private GreetingsService greetingsService = new GreetingsService(userService);

    @Before
    public void setUp() {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void testGetGreetings_morning() throws Exception {
        // specify mocked behavior
        when(userService.getFirstName(99)).thenReturn("John");
        when(userService.getLastName(99)).thenReturn("Doe");
        // invoke method under test
        String greetings = greetingsService.getGreetings(99, LocalTime.of(0, 45));
        Assert.assertEquals("Failed to get greetings!", "Good Morning, John Doe!", greetings);
    }

    @Test
    public void testGetGreetings_afternoon() throws Exception {
        // specify mocked behavior
        when(userService.getFirstName(11)).thenReturn("Jane");
        when(userService.getLastName(11)).thenReturn("Doe");
        // invoke method under test
        String greetings = greetingsService.getGreetings(11, LocalTime.of(13, 15));
        Assert.assertEquals("Failed to get greetings!", "Good Afternoon, Jane Doe!",
greetings);
    }
}
```

"Spy" para burlarse parcialmente

La anotación (o método) de @Spy se puede utilizar para simular parcialmente un objeto. Esto es útil cuando desea simular parcialmente el comportamiento de una clase. Por ejemplo, suponga que tiene una clase que utiliza dos servicios diferentes y desea burlarse solo de uno de ellos y usar la implementación real del otro servicio.

Nota al margen: aunque filosóficamente no consideraría esto como una "prueba de unidad pura" en un verdadero sentido, ya que está integrando una clase real y no está probando su clase bajo prueba en completo aislamiento. Sin embargo, esto podría ser realmente útil en el mundo real y a menudo lo uso cuando me burlo de la base de datos utilizando alguna implementación de base de datos en memoria para poder usar DAO reales.

```
Class under test:
```

```
public class GreetingsService { // class to be tested in isolation
    private UserService userService;
    private AppService appService;

    public GreetingsService(UserService userService, AppService appService) {
        this.userService = userService;
        this.appService = appService;
    }

    public String getGreetings(int userId, LocalTime time) { // the method under test
        StringBuilder greetings = new StringBuilder();
        String timeOfDay = getTimeOfDay(time.getHour());
        greetings.append("Good ").append(timeOfDay).append(", ");
        greetings.append(userService.getFirstName(userId)) // this call will be mocked
                .append(" ")
                .append(userService.getLastName(userId)) // this call will be mocked
                .append("!");
        greetings.append(" Welcome to ")
                .append(appService.getAppName()) // actual method call will be made
                .append(".");
        return greetings.toString();
    }

    private String getTimeOfDay(int hour) { // private method doesn't need to be unit tested
        if (hour >= 0 && hour < 12)
            return "Morning";
        else if (hour >= 12 && hour < 16)
            return "Afternoon";
        else if (hour >= 16 && hour < 21)
            return "Evening";
        else if (hour >= 21 && hour < 24)
            return "Night";
        else
            return null;
    }
}
```

El comportamiento de esta interfaz será burlado:

```
public interface UserService {
    String getFirstName(int userId);

    String getLastName(int userId);
}
```

Supongamos la implementación real del `UserService` :

```
public class UserServiceImpl implements UserService {
    @Override
    public String getFirstName(int userId) {
        String firstName = "";
        // some logic to get user's first name
        // this could be anything like a call to another service,
        // a database query, or a web service call
        return firstName;
    }

    @Override
    public String getLastName(int userId) {
```

```
        String lastName = "";
        // some logic to get user's last name
        // this could be anything like a call to another service,
        // a database query, or a web service call
        return lastName;
    }
}
```

El comportamiento de esta interfaz no será burlado:

```
public interface AppService {
    String getAppName();
}
```

Supongamos la implementación real de `AppService` :

```
public class AppServiceImpl implements AppService {
    @Override
    public String getAppName() {
        // assume you are reading this from properties file
        String appName = "The Amazing Application";
        return appName;
    }
}
```

Prueba de Junit con Mockito:

```
public class GreetingsServiceTest {
    @Mock
    private UserServiceImpl userService; // this class will be mocked
    @Spy
    private AppServiceImpl appService; // this class WON'T be mocked
    @InjectMocks
    private GreetingsService greetingsService = new GreetingsService(userService, appService);

    @Before
    public void setUp() {
        MockitoAnnotations.initMocks(this);
    }

    @Test
    public void testGetGreetings_morning() throws Exception {
        // specify mocked behavior
        when(userService.getFirstName(99)).thenReturn("John");
        when(userService.getLastName(99)).thenReturn("Doe");
        // invoke method under test
        String greetings = greetingsService.getGreetings(99, LocalTime.of(0, 45));
        Assert.assertEquals("Failed to get greetings!", "Good Morning, John Doe! Welcome to
The Amazing Application.", greetings);
    }

    @Test
    public void testGetGreetings_afternoon() throws Exception {
        // specify mocked behavior
        when(userService.getFirstName(11)).thenReturn("Jane");
        when(userService.getLastName(11)).thenReturn("Doe");
        // invoke method under test
```

```
        String greetings = greetingsService.getGreetings(11, LocalTime.of(13, 15));
        Assert.assertEquals("Failed to get greetings!", "Good Afternoon, Jane Doe! Welcome to
The Amazing Application.", greetings);
    }
}
```

Establecer campos privados en objetos simulados.

En su clase que está bajo prueba, puede tener algunos campos privados que no son accesibles incluso a través del constructor. En tales casos, puede utilizar la reflexión para establecer dichas propiedades. Este es un fragmento de dicha prueba JUnit.

```
@InjectMocks
private GreetingsService greetingsService = new GreetingsService(); // mocking this class

@Before
public void setUp() {
    MockitoAnnotations.initMocks(this);
    String someName = "Some Name";
    ReflectionTestUtils.setField(greetingsService, // inject into this object
        "name", // assign to this field
        someName); // object to be injected
}
```

Estoy usando de Sptring `ReflectionTestUtils.setField(Object targetObject, String name, Object value)` método aquí para simplificar, pero se puede usar el viejo y simple reflexión de Java para hacer lo mismo.

Lea Burlarse de en línea: https://riptutorial.com/es/mockito/topic/4573/burlarse-de

Capítulo 3: Clases y métodos finales simulados

Introducción

Desde Mockito 2.x tenemos la capacidad de simular clases y métodos finales.

Examples

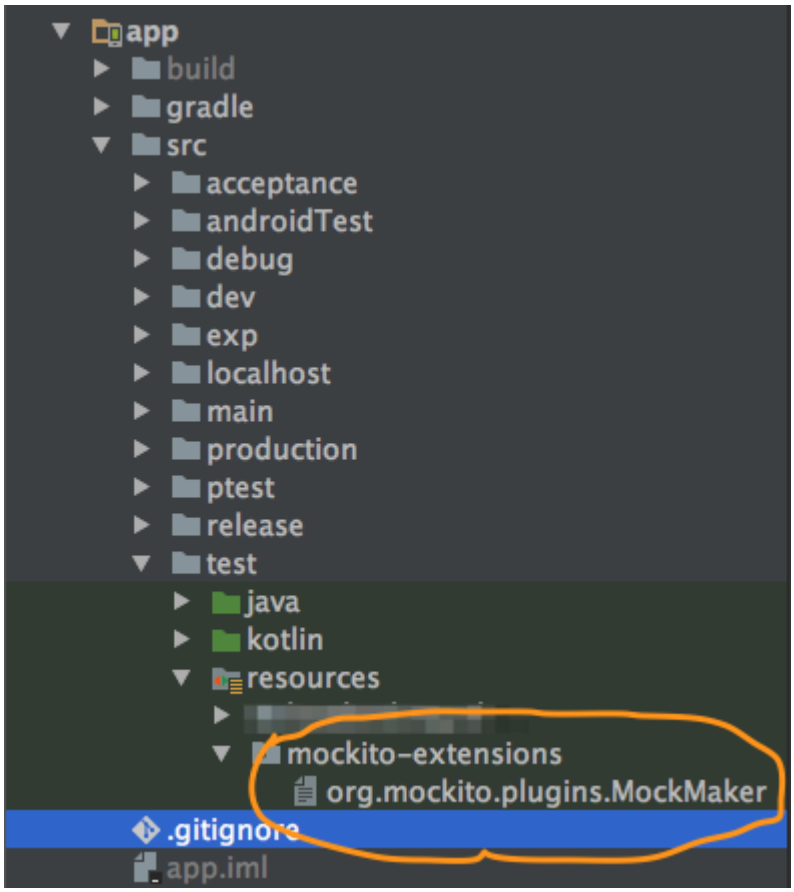
Como hacerlo

Pasos:

1. Añadir a la dependencia **Mockito** versión 2.x en su `gradle` (en el momento de escribir este texto es la última versión **2.7.22):**

```
testCompile "org.mockito:mockito-core:$versions.mockito"
```

2. Cree un archivo en los recursos de prueba con el nombre `org.mockito.plugins.MockMaker` :



3. Agregue la siguiente línea en este archivo:

```
mock-maker-inline
```

Lea Clases y métodos finales simulados en línea:
https://riptutorial.com/es/mockito/topic/9769/clases-y-metodos-finales-simulados

Capítulo 4: Mejores Prácticas de Mockito

Examples

Estilo BDDMockito

El estilo de prueba de desarrollo conducido por comportamiento (BDD) gira en torno a las etapas "dado", "cuándo" y "luego" en las pruebas. Sin embargo, Mockito clásico usa la fase "cuándo" palabra para "dada", y no incluye otras construcciones de lenguaje natural que puedan abarcar el BDD. Por lo tanto, los alias de BDDMockito se introdujeron en la versión 1.8.0 para facilitar las pruebas basadas en el comportamiento.

La situación más común es apilar los retornos de un método. En el siguiente ejemplo, el `getStudent(String)` del `StudentRepository` `new Student(givenName, givenScore)` devolverá `new Student(givenName, givenScore)` si se invoca con un argumento que es igual a `givenName` .

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

public class ScoreServiceTest {

    private StudentRepository studentRepository = mock(StudentRepository.class);

    private ScoreService objectUnderTest = new ScoreService(studentRepository);

    @Test
    public void shouldCalculateAndReturnScore() throws Exception {
        //given
        String givenName = "Johnny";
        int givenScore = 10;
        given(studentRepository.getStudent(givenName))
            .willReturn(new Student(givenName, givenScore));

        //when
        String actualScore = objectUnderTest.calculateStudentScore(givenName);

        //then
        assertEquals(givenScore, actualScore);
    }
}
```

A veces, se desea verificar si la excepción generada por la dependencia se maneja correctamente o se vuelve a derribar en un método bajo prueba. Tal comportamiento puede ser tachado en la fase "dada" de esta manera:

```
willThrow(new RuntimeException())).given(mock).getData();
```

A veces se desea establecer algunos efectos secundarios que debe introducir un método de rechazo. Especialmente puede ser útil cuando:

- El método de código auxiliar es un método que se supone que cambia el estado interno de

un objeto pasado.

- El método de código es un método vacío.

Dicho comportamiento puede ser tachado en la fase "dada" con una "Respuesta":

```
willAnswer(invocation ->
this.prepareData(invocation.getArguments()[0])).given(mock).processData();
```

Cuando se desea verificar las interacciones con un simulacro, se puede hacer en la "fase" y luego con `should()` o `should(VerificationMode)` (solo desde 1.10.5) métodos:

```
then(mock).should().getData(); // verifies that getData() was called once
then(mock).should(times(2)).processData(); // verifies that processData() was called twice
```

Cuando se desea verificar que no hubo más interacciones con un simulacro además del verificado, se puede hacer en la fase "entonces" con `shouldHaveNoMoreInteractions()` (desde 2.0.0):

```
then(mock).shouldHaveNoMoreInteractions(); // analogue of verifyNoMoreInteractions(mock) in
classical Mockito
```

Cuando se desea verificar que no hubo absolutamente ninguna interacción con un simulacro, se puede hacer en la fase "entonces" con `shouldHaveNoMoreInteractions()` (desde 2.0.0):

```
then(mock).shouldHaveZeroInteractions(); // analogue of verifyZeroInteractions(mock) in
classical Mockito
```

Cuando se desea verificar si los métodos se invocaron en orden, se puede hacer en la fase "entonces" con `should(InOrder)` (desde 1.10.5) y `should(InOrder, VerificationMode)` (desde 2.0.0):

```
InOrder inOrder = inOrder(mock);

// test body here

then(mock).should(inOrder).getData(); // the first invocation on the mock should be getData()
invocation
then(mock).should(inOrder, times(2)).processData(); // the second and third invocations on the
mock should be processData() invocations
```

Lea Mejores Prácticas de Mockito en línea: https://riptutorial.com/es/mockito/topic/4651/mejores-practicas-de-mockito

Capítulo 5: Simulacros de llamadas consecutivas a un método de retorno nulo

Introducción

Los documentos de Mockito tienen un excelente ejemplo de cómo proporcionar una secuencia de respuestas para múltiples llamadas a un simulacro. Sin embargo, no cubren cómo hacer eso para un método que devuelve vacío, aparte de señalar que los métodos para anular el vaciamiento requieren el uso de la familia de métodos do .

Observaciones

Recuerde, para los métodos que no son nulos, se prefiere la versión `when(mock.method()).thenThrow().thenReturn()` (ver documentos) porque es un argumento de tipo seguro y más legible.

Examples

Fingiendo un error transitorio

Imagine que está probando el código que realiza una llamada a esta interfaz y desea asegurarse de que el código de reintento funciona.

```
public interface DataStore {
    void save(Data data) throws IOException;
}
```

Podrías hacer algo como esto:

```
public void saveChanges_Retries_WhenDataStoreCallFails() {
    DataStore dataStore = new DataStore();
    Data data = new Data();
    doThrow(IOException.class).doNothing().when(dataStore).save(data);

    dataStore.save(data);

    verify(dataStore, times(2)).save(data);
    verifyDataWasSaved();
}
```

Lea Simulacros de llamadas consecutivas a un método de retorno nulo en línea: https://riptutorial.com/es/mockito/topic/9010/simulacros-de-llamadas-consecutivas-a-un-metodo-de-retorno-nulo

Capítulo 6: Verificar llamadas de método

Examples

Método simple de verificación de llamadas

Uno puede verificar si un método fue llamado en un simulacro usando `Mockito.verify()` .

```
Original mock = Mockito.mock(Original.class);
String param1 = "Expected param value";
int param2 = 100; // Expected param value

//Do something with mock

//Verify if mock was used properly
Mockito.verify(mock).method();
Mockito.verify(mock).methodWithParameters(param1, param2);
```

Verificar orden de llamadas

En algunos casos, puede que no sea suficiente saber si se llamaron más de un método. El orden de llamada de los métodos también es importante. En tal caso, es posible utilizar `InOrder` clase de `Mockito` para verificar el orden de los métodos.

```
SomeClass mock1 = Mockito.mock(SomeClass.class);
otherClass mock2 = Mockito.mock(OtherClass.class);

// Do something with mocks

InOrder order = Mockito.inOrder(mock1, mock2)
order.verify(mock2).firstMethod();
order.verify(mock1).otherMethod(withParam);
order.verify(mock2).secondMethod(withParam1, withParam2);
```

`InOrder.verify()` funciona igual que `Mockito.verify()` todos los demás aspectos.

Verifique los argumentos de la llamada usando ArgumentCaptor

`ArgumentCaptor` recibirá los argumentos de invocación reales que se han pasado al método.

```
ArgumentCaptor<Foo> captor = ArgumentCaptor.forClass(Foo.class);
verify(mockObj).doSomethind(captor.capture());
Foo invocationArg = captor.getValue();
//do any assertions on invocationArg
```

Para casos de invocaciones múltiples de método simulado para recibir todos los argumentos de invocación

```
List<Foo> invocationArgs = captor.getAllValues();
```

El mismo enfoque se utiliza para capturar varargs.

También existe la posibilidad de crear `ArgumentCaptor` usando la anotación `@Captor` :

```
@Captor
private ArgumentCaptor<Foo> captor;
```

Lea Verificar llamadas de método en línea: https://riptutorial.com/es/mockito/topic/4858/verificar-llamadas-de-metodo

Creditos

S. No	Capítulos	Contributors
1	Empezando con mockito	Abubakkar, Brice, cantido, Chriss, codebox, Community, Constantine, Eugen Martynov, fiskeben, gandreadis, J. Schneider, Kevin Welker, kiuby_88, Lorenzo Murrocu, Mark Rotteveel, Matsemann, nhouser9, Nicktar, Squidward, thug-gamer, Tim van der Lippe, Walery Strauch
2	Burlarse de	Nicktar, RamenChef, Suraj Bajaj
3	Clases y métodos finales simulados	Eugen Martynov
4	Mejores Prácticas de Mockito	Constantine
5	Simulacros de llamadas consecutivas a un método de retorno nulo	Cameron Stone
6	Verificar llamadas de método	Abdullah, Sergii Bishyr