Testing Efectivo en Java

Círculo Siete Capacitación

Clase 7 de 12

19 Agosto 2025

Introducción a Spock Framework

- Spock es un framework de testing para aplicaciones Java y Groovy, diseñado para ser más expresivo, legible y potente que las alternativas tradicionales como JUnit y TestNG.
- Está escrito en Groovy, pero puede probar cualquier aplicación basada en la JVM (Java, Kotlin, Scala, etc.).
- Su objetivo principal es que las pruebas sean:
  - Más fáciles de leer (casi como especificaciones).
  - Más completas (incluye mocks, stubs y spies integrados).
  - Más expresivas (usa un DSL claro y cercano al lenguaje natural).

Principales características

- Sintaxis clara y expresiva: Usa bloques como **given**, **when**, **then**, que facilitan el estilo BDD/Specification.
- Mocks integrados: No necesitas Mockito; Spock trae sus propias herramientas de mocking y stubbing.
- Data-driven testing: Permite parametrizar fácilmente pruebas con tablas o variables (where:).
- Integración con JUnit: Se ejecuta en cualquier entorno que soporte JUnit (Maven, Gradle, IDEs).
- Testing de excepciones: Sintaxis concisa para verificar excepciones esperadas.
- Extensiones y compatibilidad: Fácil de extender y combinar con librerías de prueba modernas (como Testcontainers, RestAssured, etc.).

Ejemplo sencillo

```java
public class Calculator {
    int sum(int a, int b) {
        return a + b;
    }
}
```

```groovy
// Esto es Groovy
import spock.lang.Specification

class CalculatorSpec extends Specification {

    def "should return correct sum"() {
        given: "a calculator"
            def calculator = new Calculator()

        when: "two numbers are added"
            def result = calculator.sum(2, 3)

        then: "the result should be correct"
            result == 5
    }
}
```

Características observadas

- **Specification**: clase base de Spock (similar a **@Test** en JUnit).
- **given/when/then**: estructura clara (BDD).
- Asserts naturales (==), no métodos estáticos.

Pruebas parametrizadas

```
import spock.lang.Specification

class CalculatorSpec extends Specification {

    def "sum of #a and #b should be #expected"(int a, int b, int expected) {
        given:
            def calculator = new Calculator()
        expect:
            calculator.sum(a, b) == expected
        where:
            a  |  b  || expected
            1  |  2  || 3
            3  |  5  || 8
            10 | -2  || 8
    }
}
```

Laboratorio

- Revisar CalculatorSpec.groovy

Mocking

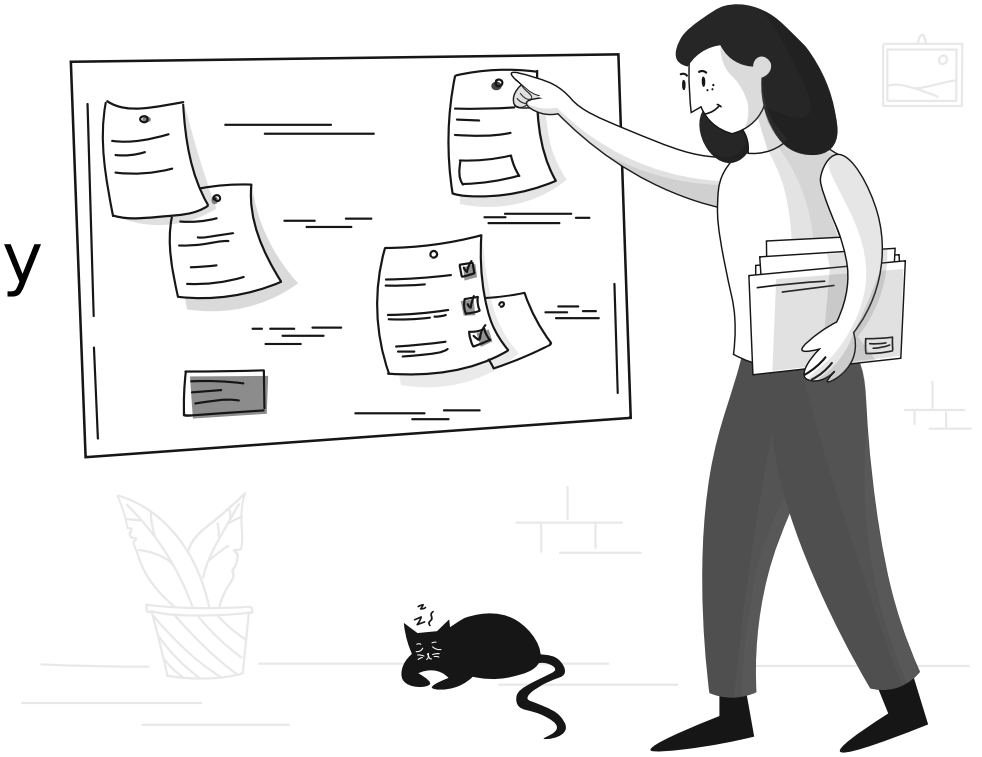
- En Spock, el mocking es una de las características más potentes y simples de usar. Se basa en el concepto de test doubles (dobles de prueba), que permiten aislar la unidad bajo prueba simulando el comportamiento de sus dependencias.
- Spock unifica en una sola sintaxis lo que en otros frameworks se divide en mocks, stubs y spies, lo que lo hace muy expresivo.

¿Cómo funciona el mocking en Spock?

- En Spock, se crean dobles con la sintaxis:
  - def dependency = Mock(DependencyClass)
- Ese objeto ahora puede:
  - Responder con valores predefinidos (stubbing).
  - Registrar cómo fue invocado (mocking).
  - Hacer ambas cosas.
- El motor de Spock intercepta las llamadas a métodos y permite:
  - Definir expectativas: cuántas veces se invoca un método, con qué parámetros.
  - Definir respuestas: qué valor devuelve o qué excepción lanza.

Laboratorio

- Revisar OrderProcessorSpec.groovy



Tipos de dobles en Spock

- Spock soporta tres variantes principales:
  - Stub
  - Mock
  - Spy

Stub

- Un stub se usa para devolver respuestas predefinidas sin verificar interacciones.
- Características:
  - Solo interesa el qué devuelve.
  - No verifica cuántas veces o con qué argumentos se invoca.

```
def dependency = Stub(DependencyClass) {
    compute(_) >> 42   // siempre devuelve 42
}
```

Mock

- Un mock combina stubbing con verificación de interacciones.
- Características:
  - Puedes declarar expectativas sobre cuántas veces debe llamarse un método.
  - Puedes combinarlo con respuestas (>>).
  - Si la expectativa no se cumple, el test falla.

```
def dependency = Mock(DependencyClass)

when:
    sut.doSomething()

then:
    1 * dependency.compute("hello") >> "world"
```

Spy

- Un spy envuelve una implementación real, permitiendo usar el código original pero también observar o modificar algunas interacciones.
- Características:
  - Ejecuta la lógica real por defecto.
  - Permite sobreescribir métodos específicos.
  - Útil para probar clases grandes sin necesidad de duplicar toda la lógica.

```groovy
def service = Spy(RealService)

when:
    service.process("data")

then:
    1 * service.helperMethod(_) >> "mocked"
```

Diferencias entre ellos

Tipo	Propósito Principal	Ejecuta lógica real	Verifica llamadas
Stub	Devolver respuestas predeterminadas	No	No
Mock	Responder y verificar interacciones	No	Si
Spy	Observar y parcialmente reemplazar lógica existente	Si (por defecto)	Si

Laboratorio

- CalculatorService (SUT) depende de un NumberRepository y de un Notifier.
- NumberRepository devuelve números desde alguna fuente.
- Notifier envía un mensaje cuando el resultado supera un umbral.
- Revisar:
  - CalculatorServiceStubSpec
  - CalculatorServiceMockSpec
  - CalculatorServiceSpySpec

Consejos y buenas prácticas

- Empieza con Stubs cuando solo necesites valores; sube a Mocks cuando te importe la interacción.
- Evita sobre-especificar interacciones (tests frágiles). Verifica solo lo relevante para el comportamiento.
- Spies son útiles para legado/lógica compleja, pero úsalos con moderación: si necesitas muchos parches, quizá el diseño necesita extraer dependencias.
- Usa predicados de argumentos y closures para respuestas dependientes de los parámetros.
- Termina con 0 * _ cuando quieras asegurar que no hubo llamadas inesperadas (gran ayuda para detectar efectos colaterales).