Icono *Training Consulting*

Formación

Curso Testing Back

2025

Formador

Ana Isabel Vegas

INGENIERA INFORMÁTICA con Master Máster Universitario en Gestión y Análisis de Grandes Volúmenes de Datos: Big Data, tiene la certificación PCEP en Lenguaje de Programación Python y la certificación JSE en Javascript. Además de las certificaciones SCJP Sun Certified Programmer for the Java 2 Platform Standard Edition, SCWD Sun Certified Web Component Developer for J2EE 5, SCBCD Sun Certified Business Component Developer for J2EE 5, SCEA Sun Certified Enterprise Architect for J2EE 5.

Desarrolladora de Aplicaciones FULLSTACK, se dedica desde hace + de 20 años a la CONSULTORÍA y FORMACIÓN en tecnologías del área de DESARROLLO y PROGRAMACIÓN.

training@iconotc.com

Testing Back

❑ **Duración:** 20 horas

❑ **Modalidad:** Presencial / Remoto

❑ **Fechas/Horario:**

- Días 13, 14, 15 y 16 Octubre 2025
- Horario de 15:00 a 20:00 hs

❑ **Contenidos:**

- Introducción
  - Conceptos: TDD, BDD
  - Tipos de Pruebas
  - Generación de Informes
- Pruebas de Unidad e integración en Java
  - Estrategia de Pruebas
  - Diseño de Pruebas
  - Pruebas de Unidad: Junit
  - Pruebas entre componentes o clases
  - Pruebas de integración: Mockito
  - Mock y Proxies
- Pruebas de Aceptación
  - CucumberJVM
  - Gherkin
  - Selenium
  - Webdriver
- Jenkins
  - Introducción
  - Integración Git
  - Integración Maven
  - Integración Pruebas
  - Pipelines
  - Sonar

Volver al índice

Introducción

Tema 1

Test Driven Development (TDD)

- Desarrollo guiado por pruebas, o Test Driven Development (TDD) es una práctica de programación que involucra otras dos prácticas:
  - Escribir las pruebas primero (Test First Development)
  - Refactorización (Refactoring).

- Para escribir las pruebas generalmente se utiliza la Prueba Unitaria

Test Driven Development (TDD)

- TDD (Test-Driven Development) es una metodología de desarrollo de software que consiste en escribir primero las pruebas unitarias antes de implementar el código de la funcionalidad. Este enfoque sigue un ciclo iterativo conocido como "rojo-verde-refactorizar":
  - **Fase roja**: Se escribe una prueba que inicialmente falla.
  - **Fase verde**: Se implementa el código necesario para que la prueba pase.
  - **Fase de refactorización**: Se optimiza el código sin cambiar su comportamiento.

- El objetivo principal de TDD es crear un código más robusto, modular y de mayor calidad. Al escribir las pruebas primero, los desarrolladores se centran en una única característica a la vez, lo que mejora la arquitectura de la solución y facilita la detección temprana de errores.

Ventajas TDD

- **Mejora la calidad del código**: Al escribir pruebas antes de implementar la funcionalidad, se asegura que cada línea de código esté justificada y cumpla con los requisitos establecidos.
- **Código más simple y menos redundante**: El ciclo constante de refactorización promueve la eliminación de código innecesario y la reutilización de funciones existentes.
- **Aumenta la productividad**: Se reduce el tiempo dedicado a la depuración, ya que los errores se detectan y corrigen tempranamente en el proceso de desarrollo.
- **Facilita la escalabilidad:** El enfoque modular del TDD permite crear código más fácil de mantener y expandir a medida que el proyecto crece.
- **Minimiza errores**: La detección temprana de problemas reduce significativamente el número de errores que llegan a producción.

Ventajas TDD

- **Mejora el diseño del software**: El proceso iterativo de TDD fomenta un diseño emergente y más efectivo de la arquitectura del software.
- **Proporciona documentación actualizada**: Los propios tests sirven como documentación ejecutable del comportamiento esperado del código.
- **Facilita la refactorización**: Permite mejorar el código existente sin temor a introducir nuevos errores, ya que las pruebas garantizan que la funcionalidad se mantiene.
- **Mejora la comunicación en el equipo**: Los problemas se abordan en períodos cortos de tiempo, fomentando la colaboración entre los miembros del equipo.
- **Aumenta la confianza en el código**: La alta cobertura de pruebas que se logra con TDD proporciona mayor seguridad sobre la funcionalidad del código en diversos escenarios.

El algoritmo TDD

- Escribir la prueba. Para escribir la prueba, el desarrollador debe entender claramente las especificaciones y los requisitos. El diseño del documento deberá cubrir todos los escenarios de prueba y condición de Excepciones.
- Escribir el código haciendo que pase la prueba. Este paso fuerza al programador a tomar la perspectiva de un cliente considerando el código a través de sus interfaces. Ésta es la parte conducida por el diseño, del TDD. Como parte de la calibración de la prueba, el código debe fallar la prueba significativamente las primeras veces.
- Ejecutar las pruebas automatizadas. Si pasan, el programador puede garantizar que el código resuelve los casos de prueba escritos. Si hay fallos, el código no resolvió los casos de prueba.
- Refactorización y limpieza en el código. Después se vuelven a efectuar los casos de prueba y se observan los resultados.
- Repetición. Después se repetirá el ciclo y se comenzará a agregar las funcionalidades adicionales o a arreglar cualquier error.

Behavior Driven Development (BDD)

- BDD en testing significa "Behavior Driven Development" o "Desarrollo guiado por comportamiento" y es una metodología ágil que busca mejorar la colaboración entre desarrolladores, testers y el área de negocio usando pruebas escritas en lenguaje natural.

- BDD es un proceso de desarrollo de software que surge como evolución del TDD (Test Driven Development). Mientras que el TDD se enfoca en pruebas unitarias para las funcionalidades técnicas, el BDD redefine las pruebas poniéndolas en términos de comportamientos esperados desde la perspectiva del usuario o cliente final. Esto significa que las pruebas se describen usando ejemplos reales y lenguaje comprensible para todos los involucrados en el proyecto

BDD vs TDD

- BDD se orienta al comportamiento del sistema y a la experiencia del usuario, mientras que TDD se enfoca exclusivamente en pruebas técnicas de código. Ambas metodologías pueden coexistir, pero BDD fomenta mayor integración y participación de todos los equipos, no sólo los desarrolladores.

- Un escenario en BDD podría ser:
  - Dado que el usuario no ha introducido ningún dato en un formulario
  - Cuando hace clic en "Enviar"
  - Entonces debe mostrar un mensaje de validación.

- Esta estructura facilita que todos los miembros del equipo comprendan qué se espera del sistema ante determinadas acciones.

Tipos de pruebas

- Pruebas unitarias
- Pruebas de integración
- Pruebas de sistema
- Pruebas funcionales
- Pruebas de carga
- Pruebas de estrés
- Pruebas de aceptación

Pruebas unitarias

- Con ellas probamos las unidades del software, normalmente métodos.
- Por ejemplo, escribimos estas pruebas para comprobar si un método de una clase funciona correctamente de forma aislada.
- Por ejemplo, aunque estés probando un método que realice una serie de cosas y al final envíe un correo electrónico a través de un servidor de correo, en una prueba unitaria no tienes que probar que el correo se ha mandado correctamente.
- Buenas pruebas unitarias no irían contra una base de datos, por ejemplo, sino que simularían la conexión.

Pruebas de integración

- En este caso probamos cómo es la interacción entre dos o mas unidades del software.
- Este tipo de pruebas verifican que los componentes de la aplicación funcionan correctamente actuando en conjunto.
- Siguiendo con el caso anterior, las pruebas de integración son las que comprobarían que se ha mandado un email, la conexión real con la base de datos etc.
- Este tipo de pruebas son dependientes del entorno en el que se ejecutan. Si fallan, puede ser porque el código esté bien, pero haya un cambio en el entorno.

Pruebas de sistema

- Lo ideal sería realizar este tipo de pruebas después de las pruebas de integración.
- Aquí se engloban tipos de pruebas cuyo objetivo es probar todo el sistema software completo e integrado, normalmente desde el punto de vista de requisitos de la aplicación.
- Aquí aparecerían las pruebas funcionales, pruebas de carga, de estrés etc

Pruebas funcionales

- En este caso, el objetivo de las pruebas funcionales es comprobar que el software que se ha creado cumple con la función para la que se había pensado.
- En este tipo de pruebas lo que miramos, lo que nos importan, son las entradas y salidas al software. Es decir, si ante una serie de entradas el software devuelve los resultados que nosotros esperábamos.
- Aquí solo observamos que se cumpla la funcionalidad, no comprobamos que el software esté bien hecho, no miramos el diseño del software. Estudiamos el software desde la perspectiva del cliente, no del desarrollador.
- Por eso este tipo de pruebas entran dentro de lo que se llaman pruebas de caja negra: aquí no nos centramos en cómo se generan las respuestas del sistema, solo analizamos los datos de entrada y los resultados obtenidos.
- Una prueba funcional podría ser por ejemplo comprobar que en el formulario de mi página web si relleno un campo de teléfono con zzzz muestre un mensaje de campo no válido, y no me deje registrar.
- Estas pruebas pueden ser manuales, o automatizarse con herramientas. Una de las más conocidas para web es Selenium, en sus distintas variantes.

Pruebas de carga

- Las pruebas de carga son un tipo de prueba de rendimiento del sistema. Con ellas observamos la respuesta de la aplicación ante un determinado número de peticiones.

- Aquí entraría por ejemplo ver cómo se comporta el sistema ante X usuarios que entran concurrentemente a la aplicación y realizan ciertas transacciones.

Pruebas de estrés

- Este es otro tipo de prueba de rendimiento del sistema.
- El objetivo de estas pruebas es someter al software a situaciones extremas, intentar que el sistema se caiga, para ver cómo se comporta, si es capaz de recuperarse o tratar correctamente un error grave.

Pruebas de aceptación

- Por último, las pruebas de aceptación se realizan para comprobar si el software cumple con las expectativas del cliente, con lo que el cliente realmente pidió.

Generación de informes

- La generación de informes en pruebas es el proceso de documentar y presentar los resultados obtenidos durante la validación de software, facilitando análisis, toma de decisiones y comunicación entre equipos involucrados.
- Los informes de pruebas permiten obtener información crítica sobre la calidad del sistema, identificar riesgos, analizar defectos y registrar las condiciones específicas en que cada prueba fue ejecutada.
- Estos documentos son indispensables para clientes, líderes técnicos y desarrolladores ya que ayudan a mejorar el producto y a tomar decisiones apoyadas en datos concretos.

Elementos principales del informe

- Un informe de pruebas suele incluir:
  - Resumen ejecutivo (descripción general y conclusiones clave).
  - Índice de contenidos (para accesibilidad).
  - Detalle de lo que fue testeado y lo que no (áreas y razones).
  - Resultados detallados de cada caso (pasaron/fallaron/pending bugs).
  - Evaluación de defectos, riesgos y recomendaciones.
  - Conclusiones y posibles acciones de mejora.

Herramientas y automatización

- Actualmente existen herramientas como **Allure Report** que permiten la generación automática, visual y colaborativa de informes de pruebas integrados en pipelines de CI/CD.
- Estas herramientas recopilan datos, organizan resultados, agregan pruebas fallidas y exitosas, y permiten compartir el progreso con todo el equipo.
- Muchas de ellas soportan pruebas unitarias, de API y de extremo a extremo. Además, suelen ser independientes del lenguaje y ayudan a identificar regresiones y rastrear fallos históricos.

Buenas prácticas

- Es fundamental que los informes sean claros, breves y fáciles de leer, evitando la documentación excesiva cuando no agrega valor, especialmente en entornos ágiles.
- Deben detallar de manera precisa los defectos y escenarios pendientes, y ser actualizados y revisados constantemente.
- En resumen, la generación de informes en pruebas implica documentar y compartir de forma estructurada los resultados, defectos y recomendaciones, utilizando herramientas y buenas prácticas para facilitar el trabajo colaborativo y la mejora continua del software.

Pruebas de Unidad e integración en Java

Tema 2

Prueba Unitaria

- Una prueba unitaria es una forma de probar el correcto funcionamiento de un módulo de código. Esto sirve para asegurar que cada uno de los módulos funcione correctamente por separado.

- La idea es escribir casos de prueba para cada función no trivial o método en el módulo de forma que cada caso sea independiente del resto. Esto último es la esencia de una prueba unitaria: se prueba al componente de forma aislada a todos los demás.

Características de las pruebas unitarias

- Para que una prueba unitaria sea buena se deben cumplir los siguientes requisitos:
  - **Automatizable**: no debería requerirse una intervención manual. Esto es especialmente útil para Integracion Continua.
  - **Completas**: deben cubrir la mayor cantidad de código.
  - **Repetibles o Reutilizables**: no se deben crear pruebas que sólo puedan ser ejecutadas una sola vez. También es útil para Integracion Continua.
  - **Independientes**: la ejecución de una prueba no debe afectar a la ejecución de otra.
  - **Profesionales**: las pruebas deben ser consideradas igual que el código, con la misma profesionalidad, documentación, etc.

Ventajas en el uso de Pruebas Unitarias

- **Detección temprana de errores**: Identifican problemas en etapas iniciales del desarrollo.
- **Facilitan el mantenimiento**: Aseguran que los cambios en el código no rompan funcionalidades existentes.
- **Documentación viviente**: Sirven como una guía clara sobre cómo debería comportarse cada unidad.

Pasos para realizar una Prueba Unitaria

1. **Comprender el código a probar**: Identifica la unidad de código (función, método o clase) que será evaluada y define los requisitos esperados de su comportamiento.

2. **Configurar el entorno de prueba**:
   - Prepara los datos necesarios para la prueba.
   - Configura cualquier dependencia o simulación (stubs o mocks) si la unidad depende de otros componentes.

3. **Escribir el caso de prueba**:
   - Sigue el patrón Arrange, Act, Assert (Organizar, Actuar, Afirmar):
     - Arrange: Configura los objetos y datos necesarios.
     - Act: Ejecuta la unidad de código con las entradas definidas.
     - Assert: Verifica que el resultado obtenido coincida con el esperado.

Pasos para realizar una Prueba Unitaria

4. **Ejecutar la prueba**:
   - Usa herramientas o frameworks como JUnit (Java), pytest (Python), NUnit (.NET), entre otros, para ejecutar las pruebas automáticamente.
5. **Analizar los resultados**:
   - Si la prueba falla, revisa el código o los datos de entrada para identificar y corregir errores.
   - Si pasa, documenta el caso como validado.
6. **Refactorizar y repetir**:
   - Realiza mejoras en el código si es necesario y vuelve a ejecutar las pruebas para asegurarte de que no se introdujeron errores nuevos.
7. **Automatización continua**:
   - Integra las pruebas unitarias en un pipeline de integración continua para ejecutarlas automáticamente con cada cambio en el código.

JUnit

- JUnit es un conjunto de clases (framework) que permite realizar la ejecución de clases Java de manera controlada, para poder evaluar si el funcionamiento de cada uno de los métodos de la clase se comporta como se espera.
- En función de algún valor de entrada se evalúa el valor de retorno esperado
- Si la clase cumple con la especificación, entonces JUnit devolverá que el método de la clase pasó exitosamente la prueba
- En caso de que el valor esperado sea diferente al que regresó el método durante la ejecución, JUnit devolverá un fallo en el método correspondiente

Ejecución de JUnit

- Hasta Versión **JUnit3:**
  - Definir clases que hereden de la clase junit.framework.TestCase de Junit.
  - Implementar métodos testXXX() (sin parámetros), que serán los que hagan las pruebas necesarias.
  - Inicialización del test en el método setUp() para colocar dicho código que se ejecutará antes de realizar cada prueba.
  - Terminación del test en el método tearDown() en el caso de que queramos liberar algunos recursos después de cada prueba.
  - Clases AssertXXX para comprobación de los resultados: AssertTrue
  - Lanzamiento de conjunto de tests mediante método suite():

Ejecución de JUnit

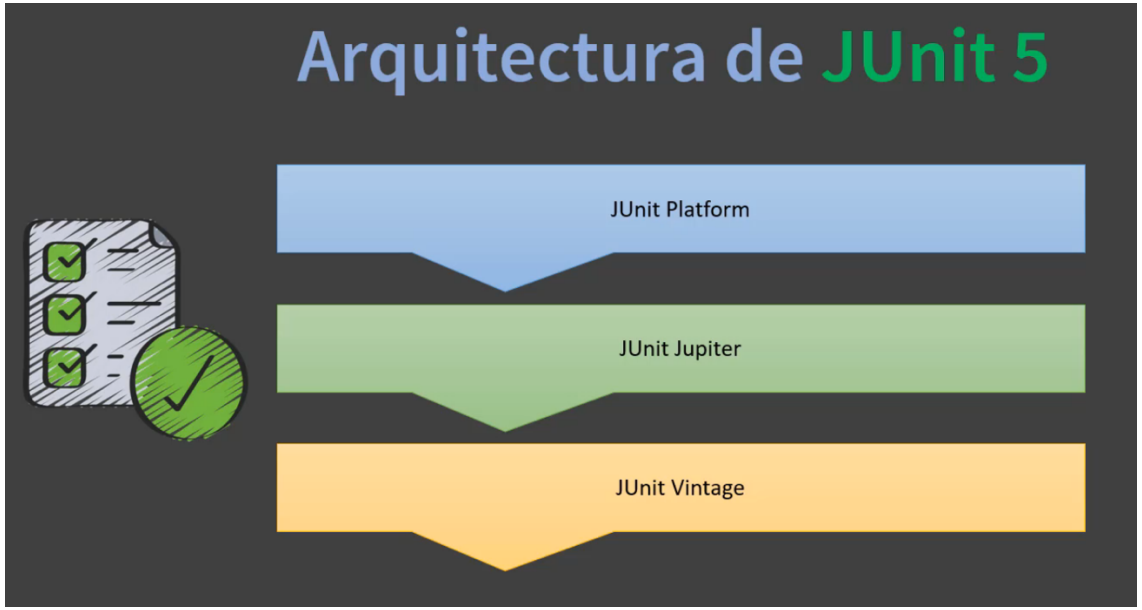
- Versión **JUnit4:**
  - Incluye anotaciones (Java 5 annotations) en lugar de utilizar herencia:
  - @Test sustituye a la herencia de TestCase.
  - @Before y @After como sustitutos de setUp y tearDown.
  - @BeforeClass y @AfterClass, metodos estáticos para inicializar y liberar recursos en el Test

Ejecución de JUnit

- Versión **JUnit5:**
  - Permite trabajar con todas las novedades incluidas en Java 8:
    - Programación funcional
    - Lambdas
    - Streams, …etc.

Arquitectura Junit 5

- JUnit 5 se compone de tres subproyectos principales que forman su arquitectura:



Arquitectura Junit 5

1. **JUnit Platform**: Es la base de JUnit 5 y sirve como fundamento para lanzar marcos de prueba en la JVM. Sus funciones principales incluyen:
   - Definir la API TestEngine para desarrollar marcos de prueba.
   - Proporcionar un Lanzador de Consola para ejecutar pruebas desde la línea de comandos.
   - Ofrecer el JUnit Platform Suite Engine para ejecutar conjuntos de pruebas personalizados.
   - Brindar soporte en IDEs populares y herramientas de compilación.

2. **JUnit Jupiter**: Es el nuevo modelo de programación y extensión para escribir pruebas y extensiones en JUnit. Incluye:
   - Nuevas anotaciones como @TestFactory, @DisplayName, @Nested, @Tag, @ExtendWith, @BeforeEach, @AfterEach, y @BeforeAll.
   - Soporte para pruebas parametrizadas y dinámicas.

3. **JUnit Vintage**: Proporciona un motor de prueba para ejecutar pruebas basadas en JUnit 3 y JUnit 4 en la plataforma.

Dependencia Junit 5

```xml
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.6.3</version>
    </dependency>
</dependencies>
```

Principales anotaciones en Junit 5

- **@Test**: Indica que un método es un caso de prueba.
- **@DisplayName**: Permite definir un nombre personalizado para una clase de prueba o un método de prueba, mejorando la legibilidad de los resultados.
- **@BeforeEach**: Indica que el método anotado se ejecutará antes de cada método de prueba (reemplaza a @Before de JUnit 4).
- **@AfterEach**: Indica que el método anotado se ejecutará después de cada método de prueba (reemplaza a @After de JUnit 4).
- **@BeforeAll**: Indica que el método anotado se ejecutará una vez antes de todos los métodos de prueba en la clase (reemplaza a @BeforeClass).
- **@AfterAll**: Indica que el método anotado se ejecutará una vez después de todos los métodos de prueba en la clase (reemplaza a @AfterClass).

Principales anotaciones en Junit 5

- **@Nested**: Indica que la clase anotada es una clase de prueba anidada y no estática.
- **@Tag**: Declara etiquetas para filtrar pruebas.
- **@Disabled**: Desactiva temporalmente una prueba (reemplaza a @Ignore de JUnit 4).
- **@TestFactory**: Denota un método que es una fábrica de pruebas para pruebas dinámicas.
- **@ParameterizedTest**: Indica que un método es una prueba parametrizada.
- **@RepeatedTest**: Permite crear pruebas que se ejecutan varias veces de forma automática.

Principales métodos de Aserciones en Junit 5

- En JUnit 5, los métodos de aserción son fundamentales para validar los resultados de las pruebas unitarias. Estos métodos están disponibles en la clase estática org.junit.jupiter.api.Assertions. A continuación, se describen los principales métodos de aserción:

1. **assertEquals(expected, actual):**
    - Verifica que el valor esperado sea igual al valor actual.
    - Soporta un tercer argumento opcional para un mensaje personalizado y un parámetro delta para comparaciones con valores decimales.

```
assertEquals(5, 2 + 3, "Los valores no son iguales");
```

2. **assertNotEquals(expected, actual):**
    - Verifica que dos valores sean diferentes.

Principales métodos de Aserciones en Junit 5

3. **assertTrue(condition) / assertFalse(condition):**
    - Verifica que una condición sea verdadera o falsa respectivamente.

```
assertTrue(5 > 3, "La condición no es verdadera");
```

4. **assertNull(object) / assertNotNull(object):**
    - Verifica que un objeto sea nulo o no nulo.

```
assertNotNull(myObject, "El objeto no debe ser nulo");
```

Principales métodos de Aserciones en Junit 5

5. **assertAll(heading, executable...):**
   - Agrupa múltiples aserciones y ejecuta todas, incluso si alguna falla.

```
assertAll("Validaciones",
    () -> assertEquals(10, 5 + 5),
    () -> assertNotNull("Texto")
);
```

6. **assertThrows(expectedType, executable):**
   - Verifica que se lance una excepción específica durante la ejecución de un bloque de código.

```
assertThrows(IllegalArgumentException.class, () -> {
    throw new IllegalArgumentException("Error");
});
```

Principales métodos de Aserciones en Junit 5

7. **assertIterableEquals(expected, actual):**
    - Compara dos objetos Iterable para verificar que sean iguales.

8. **assertArrayEquals(expectedArray, actualArray):**
    - Compara dos arreglos para verificar que sean iguales.

9. **assertLinesMatch(expectedLines, actualLines):**
    - Verifica que dos listas de cadenas coincidan línea por línea.

10. **fail(message):**
    - Marca explícitamente una prueba como fallida con un mensaje personalizado.

Prueba Unitaria

- Para funcionar, un test unitario no debería utilizar ningún framework de aplicación ni requerir una dependencia externa: ni Spring, ni Struts, ni una base de datos, ni un servidor de aplicaciones, ni un EJB desplegado, ni ningún otro servicio cualquiera funcionando.

- Así, el testeo unitario se encarga de probar el funcionamiento aislado de la clase. Todas las dependencias que tenga la clase bajo test deberían ser simuladas usando distintos Mock Object.

Mock Object

- Un Mock Object es un "objeto falso", un objeto que representa a otro y lo sustituye en funcionalidad. Este patrón es utilizado ampliamente en la Prueba Unitaria para asegurar un correcto aislamiento de la clase bajo test.
- Así, las dependencias que tenga nuestro objeto a testear pueden ser reemplazadas por mocks que funcionen como nosotros queremos. De lograr esto, podremos testear en forma aislada a nuestra clase, sin preocuparnos por sus dependencias (más aún, sin preocuparnos por si realmente funcionan estas dependencias).
- Usando Mock Objects podemos asegurar un "entorno perfecto y a medida", haciendo que este entorno responda como nosotros necesitamos. Luego, si el test de la clase falla, será por un problema en esta misma clase (y no en sus dependencias ya que, por hipótesis, el entorno era ideal).

Frameworks de Mocks

- Existen varios frameworks que ayudan a la creación de mocks:
  - EasyMock
  - Mockito
  - MockEjb

- Mockito es uno de los más conocidos.
- EasyMock es el usado por el equipo de Spring para testear su framework.

Mockito

- Mockito es una librería Java para la creación de Mock Object muy usados para el testeo unitario
- Mockito fue creado con el objetivo de simplificar y solucionar algunos de los temas antes mencionados.
- EasyMock y Mockito puede hacer exactamente las mismas cosas, pero Mockito tiene un API más natural y práctico de usar.

Dependencias para usar Mockito

```xml
<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter</artifactId>
        <version>5.6.3</version>
    </dependency>
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-core</artifactId>
        <version>3.6.28</version>
    </dependency>
    <dependency>
        <groupId>org.mockito</groupId>
        <artifactId>mockito-junit-jupiter</artifactId>
        <version>3.6.28</version>
    </dependency>
```

Principales anotaciones de Mockito

- Mockito es un framework de pruebas unitarias en Java que permite simular objetos y comportamientos. Sus anotaciones simplifican la creación, configuración e inyección de mocks en los tests. A continuación, se describen las principales anotaciones:
- **@Mock:**
  - Crea un objeto simulado (mock) de una clase o interfaz.
  - Se utiliza para reemplazar dependencias reales con simulaciones en las pruebas.

```
@Mock
private UsuarioRepositorio usuarioRepositorio;
```

Principales anotaciones de Mockito

- **@InjectMock:**
  - Crea una instancia del objeto que se está probando e inyecta automáticamente los mocks creados con @Mock o @Spy en sus dependencias.

```
@InjectMocks
private UsuarioServicio servicio;
```

- **@Spy:**
  - Envuelve un objeto real para espiar su comportamiento, permitiendo llamar a métodos reales y simular otros.

```
@Spy
private List<String> lista = new ArrayList<>();
```

Principales anotaciones de Mockito

- **@Captor:**
  - Crea un captor para capturar argumentos pasados a métodos simulados.

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

- **@MockBean:**
  - Reemplaza un bean real por un mock dentro del contexto de Spring.

```
@MockBean
private Repositorio repositorioMock;
```

Principales anotaciones de Mockito

- **@ExtendWith(MockitoExtension.class):**
  - Habilita el uso de anotaciones de Mockito en JUnit 5, inicializando automáticamente los mocks y spies.

```
@ExtendWith(MockitoExtension.class)
class MiTest {
    @Mock
    private MiDependencia dependencia;
}
```

Principales métodos de Mockito

- Mockito ofrece una variedad de métodos para crear y manejar objetos simulados (mocks) en pruebas unitarias. A continuación, se describen los métodos más importantes:
- **mock(Class<T> classToMock):**
    - Crea un mock de una clase o interfaz específica.

```
List<String> mockList = Mockito.mock(List.class);
```

- **when(T methodCall):**
    - Define el comportamiento esperado de un método simulado.

```
Mockito.when(mockList.size()).thenReturn(5);
```

Principales métodos de Mockito

- **thenReturn(value):**
  - Especifica el valor que debe devolverse cuando se llama al método simulado.

```
Mockito.when(mockList.get(0)).thenReturn("Hola");
```

- **verify(mock).method():**
  - Verifica si un método específico fue llamado en el mock, con los argumentos esperados.

```
Mockito.verify(mockList).add("Elemento");
```

- **doReturn(value).when(mock).method():**
  - Similar a when(), pero se utiliza cuando el método simulado no puede ser interceptado directamente (por ejemplo, métodos void).

```
Mockito.doReturn("Hola").when(mockList).get(0);
```

Principales métodos de Mockito

- **doNothing().when(mock).method():**
  - Define que no se haga nada cuando se llame a un método void.

```
Mockito.doNothing().when(mockList).clear();
```

- **doThrow(exception).when(mock).method():**
  - Configura un mock para lanzar una excepción específica cuando se llama a un método.

```
Mockito.doThrow(new RuntimeException()).when(mockList).clear();
```

- **assertThatThrownBy(() -> ...) (junto con assertThrows):**
  - Verifica que un método lance una excepción esperada.

Principales métodos de Mockito

- **reset(mock):**
  - Restablece un mock para eliminar configuraciones previas.

```
Mockito.reset(mockList);
```

- **spy(T object):**
  - Crea un objeto espía que permite llamar a métodos reales y simular otros.

```
List<String> spyList = Mockito.spy(new ArrayList<>());
```

- **times(n) / atLeast(n) / never() (usados con verify):**
  - Verifican la cantidad de veces que un método fue invocado.

```
Mockito.verify(mockList, Mockito.times(2)).add("Elemento");
```

Pruebas de Aceptación

Tema 3

Gherkin

- Gherkin es un lenguaje sencillo, utilizado en testing para describir escenarios de prueba de manera que cualquier persona (técnica o no técnica) los pueda entender. Es el estándar para herramientas BDD como Cucumber y permite definir comportamientos esperados del sistema.

- Elementos clave de Gherkin
  - **Feature**: Describe la funcionalidad o conjunto de pruebas que se van a validar.
  - **Scenario**: Es el caso de prueba individual, que representa un comportamiento específico.
  - **Given**: Describe el estado inicial necesario (Dado).
  - **When**: Indica la acción que realiza el usuario (Cuando).
  - **Then**: Representa el resultado esperado tras la acción (Entonces).
  - **And/But**: Permite encadenar varios pasos con conectores (Y/Pero).

Gherkin

- Buenas prácticas:
  - Un escenario debe ser breve y fácil de entender.
  - Usa el patrón Given-When-Then siempre en ese orden.
  - Si tienes acciones o resultados adicionales, usa And y But para clarificar.
  - Escribe los escenarios como si fueras el usuario: foca en lo que hace y espera, no en detalles técnicos.

Cucumber JVM

- CucumberJVM es una herramienta para automatizar pruebas basadas en la metodología BDD (Behavior Driven Development), permitiendo escribir los escenarios en lenguaje natural (español o inglés) y vincularlos a código Java para validarlos automáticamente.

- ¿Cómo funciona CucumberJVM?
  - Los requisitos o funcionalidades esperadas se escriben en lenguaje Gherkin, usando archivos .feature.
  - Cada paso del escenario se vincula a un método Java concreto en las step definitions (definiciones de pasos).
  - Esto facilita la colaboración entre equipos técnicos y de negocio, ya que todos pueden leer y entender las pruebas.

Ejemplo Cucumber JVM

- Escenario en Gherkin (Suma.feature)

```
Feature: Suma de dos números
  Scenario: Usuario suma dos números
    Given que el usuario ingresa el número 4
    And ingresa el número 5
    When pulsa el botón sumar
    Then el resultado debe ser 9
```

- Definición de pasos en Java (StepDefinitions.java)

```java
import io.cucumber.java.es.Dado;
import io.cucumber.java.es.Cuando;
import io.cucumber.java.es.Entonces;
import static org.junit.Assert.*;

public class StepDefinitions {
    int numero1, numero2, resultado;

    @Dado("que el usuario ingresa el número {int}")
    public void que_el_usuario_ingresa_el_número(Integer numero) {
        if (numero1 == 0) {
            numero1 = numero;
        } else {
            numero2 = numero;
        }
    }

    @Cuando("pulsa el botón sumar")
    public void pulsa_el_boton_sumar() {
        resultado = numero1 + numero2;
    }

    @Entonces("el resultado debe ser {int}")
    public void el_resultado_debe_ser(Integer esperado) {
        assertEquals(esperado.intValue(), resultado);
    }
}
```

Webdriver

- El WebDriver es la pieza clave en Selenium para automatizar navegadores web desde código. Es una interfaz que permite controlar de forma programada un navegador, simulando acciones como abrir páginas, escribir texto, hacer clic, etc.

- ¿Qué es WebDriver?
  - Es una API que maneja distintas implementaciones específicas para cada navegador (ChromeDriver, GeckoDriver para Firefox, etc.).
  - Facilita la interacción con elementos de la página web usando selectores.
  - Funciona como puente entre tu código (Java, Python, etc.) y el navegador.

Conceptos clave de Webdriver

Concepto	Descripción
WebDriver	Interfaz para controlar el navegador
Driver específico	Programa que controla navegador concreto (p.ej. ChromeDriver)
Elemento Web	Cualquier componente HTML de la página
Selectores	Formas de encontrar elementos (por ID, nombre, clase, etiquetas...)

Pasos básicos para usar Webdriver

- Descargar Driver específico de tu navegador (p.ej. ChromeDriver para Chrome).
- Configurar la ruta al driver en tu proyecto.
- Crear una instancia de WebDriver para controlar el navegador.
- Navegar a una URL con .get("url").
- Buscar elementos de la página con métodos como .findElement(By.id("...")).
- Interactuar con esos elementos (escribir, hacer clic, leer texto).
- Cerrar el navegador con .quit() para liberar recursos.

Selenium

- ¿Qué es Selenium y para qué sirve?
  - Selenium es una herramienta de automatización de pruebas para aplicaciones web. Permite controlar navegadores (como Chrome, Firefox, Edge) mediante código, simulando acciones de un usuario real: abrir páginas, escribir textos, hacer clic, etc.
  - Automatiza pruebas repetitivas de interfaces web, asegurando que una web funciona correctamente tras cambios.
  - Es multiplataforma y compatible con muchos lenguajes (Python, Java, etc.).

Selenium

- Como agregar Selenium a tu proyecto?

```
<dependencies>
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-java</artifactId>
        <version>4.10.0</version>
    </dependency>
</dependencies>
```

- Descarga el WebDriver del navegador
- Por ejemplo, para Chrome, descarga el ChromeDriver y pon el ejecutable en una ruta accesible.

https://sites.google.com/chromium.org/driver/

Selenium

```java
public class SeleniumDemo {
    public static void main(String[] args) {
        // Configura la ruta al chromedriver
        System.setProperty("webdriver.chrome.driver", "C:/ruta/a/chromedriver.exe");

        // Crea instancia del navegador
        WebDriver driver = new ChromeDriver();

        // Abre Google
        driver.get("https://www.google.com");

        // Busca la caja de texto y escribe "Selenium"
        WebElement searchBox = driver.findElement(By.name("q"));
        searchBox.sendKeys("Selenium");
        searchBox.submit();

        // Espera unos segundos para ver resultados
        try {
            Thread.sleep(3000);
        } catch (InterruptedException e) {
            e.printStackTrace();
        }

        // Muestra el título de la página
        System.out.println("Título actual: " + driver.getTitle());

        // Cierra el navegador
        driver.quit();
    }
}
```

- **System.setProperty** establece el path del driver del navegador.
- **new ChromeDriver()** abre una instancia de Chrome controlable.
- **driver.get(url)** abre una página web.
- **findElement(By.name("q"))** busca un elemento por su atributo name.
- **sendKeys y submit** simulan interacción del usuario.
- **Thread.sleep** espera para que cargue la página (para pruebas rápidas).
- **driver.getTitle()** obtiene el título de la ventana actual.
- **driver.quit()** cierra el navegador.

Jenkins

Tema 4

Jenkins

- Jenkins es una herramienta open source escrita en Java que permite automatizar tareas repetitivas dentro del desarrollo de software, especialmente enfocada en la Integración Continua (CI) y Entrega Continua (CD). Facilita la construcción, prueba y despliegue automático de proyectos, detectando errores tempranamente y mejorando la calidad y velocidad de entrega.

- ¿Qué es Jenkins?
  - Un servidor de automatización para CI/CD.
  - Utiliza una arquitectura de servidor y nodos para distribuir tareas.
  - Permite definir trabajos (jobs) y pipelines para automatizar flujos completos.
  - Extensible con miles de plugins que integran Git, Docker, Kubernetes y más.

Jenkins

- **Instalación en Windows**
  - Descarga el instalador oficial de Jenkins (.msi) desde jenkins.io.
  - Ejecuta el instalador y sigue el asistente (asegúrate de tener Java instalado, Jenkins requiere Java).
  - El instalador inicia Jenkins como un servicio de Windows.
  - Accede a Jenkins abriendo el navegador en http://localhost:8080.
  - Copia la contraseña inicial que aparecerá en el archivo indicado para desbloquear Jenkins.
  - Sigue el asistente web para instalar plugins recomendados y crear el primer usuario.

- Si tienes Docker instalado, puedes ejecutar Jenkins en un contenedor:

```
docker run -d -p 8080:8080 -p 50000:50000 --name jenkins jenkins/jenkins:lts
```

Integrar Jenkins con Git

Paso	Acción
Instalar plugin Git	Administrar Jenkins > Plugins > Git plugin
Crear job	Nuevo Item > Proyecto estilo libre
Configurar Git	Gestión código fuente > Git URL + credenciales
Configurar disparadores	Poll SCM o webhooks para ejecuciones automáticas
Definir pasos de construcción	Scripts o comandos para build/test
Configurar webhook en GitHub	Para notificaciones inmediatas

Integrar Jenkins con Maven

- Prerrequisitos:
  - Jenkins instalado y corriendo.
  - Plugin Git instalado en Jenkins (como en pasos anteriores).
  - Maven instalado en la máquina donde corre Jenkins o configurado en Jenkins.
  - Tener un proyecto Java con archivo pom.xml en un repositorio Git.

Paso	Acción
Configurar Maven Jenkins	En Configuración global de herramientas
Crear Job	Nuevo Item > Proyecto estilo libre
Añadir Git	URL repositorio + credenciales
Añadir construcción Maven	Paso construir con objetivo `clean install`
Configurar disparadores	Poll SCM o webhook Git
Ejecutar build	Construir ahora y revisar logs

Integrar Jenkins con pruebas

- Para integrar Jenkins con pruebas automáticas en un proyecto Maven usando Git, debes configurar que Jenkins ejecute las pruebas como parte del proceso de construcción. Maven ejecuta pruebas automáticamente con el comando mvn test o dentro del ciclo mvn clean install

Paso	Acción
Ejecutar pruebas con Maven	Usar `clean test` o `clean install`
Publicar resultados en Jenkins	Post-construcción > Publicar resultados JUnit
Configurar path reportes	`**/target/surefire-reports/*.xml`
Visualizar en Jenkins	Métricas y detalles de pruebas pasadas/ fallidas

Pipelines en Jenkins

- Los pipelines en Jenkins son un conjunto de pasos o etapas definidas en código que permiten automatizar todo el ciclo de vida de la construcción, prueba y despliegue de software. Son flujos de trabajo codificados que facilitan la creación, visualización y control de procesos complejos de integración y entrega continua (CI/CD).

- Qué son pipelines en Jenkins
  - Definen procesos automatizados complejos, como: compilar código, ejecutar pruebas, análisis de calidad, despliegues, notificaciones.
  - Están codificados usualmente en un archivo llamado Jenkinsfile, que puede estar versionado junto con el código en Git.
  - Permiten mayor control, modularidad, y reproducibilidad en el proceso de DevOps.
  - Pueden ser declarativos (más estructurados) o scripted (más flexibles).

Crear Pipelines en Jenkins

Paso	Descripción
Configurar Maven	En herramientas globales de Jenkins
Preparar Jenkinsfile	Definir etapas checkout y build con Maven
Crear pipeline	Nuevo item > Pipeline > pipeline script from SCM
Ejecutar pipeline	Construir ahora y revisar ejecución

Sonar

- Integrar SonarQube con Jenkins es una práctica común para analizar la calidad del código automáticamente en pipelines. Aquí tienes una guía básica para usar SonarQube junto con Jenkins y Maven.

- Qué es SonarQube?
  - Plataforma para análisis estático que detecta bugs, vulnerabilidades y code smells.
  - Genera reportes completos de calidad del código.
  - Se integra con muchas herramientas CI/CD, incluyendo Jenkins.

Instalar SonarQube

- Requisitos previos
  - Java 11 o superior instalado en tu equipo (SonarQube lo necesita para funcionar).
  - Al menos 2 GB de memoria RAM recomendada.

- **Paso 1: Descargar SonarQube**
  - Ve a la página oficial de descargas: https://www.sonarqube.org/downloads/
  - Descarga la versión Community (gratuita) o la que necesites.
  - Descomprime el archivo descargado en la ruta donde quieras instalar SonarQube

Instalar SonarQube

- **Paso 2: Configurar SonarQube**

1. Entra a la carpeta conf dentro de la instalación:

   cd /opt/sonarqube/conf

2. Edita sonar.properties para configurar opciones como el puerto o la base de datos (por defecto usa una base de datos embebida para pruebas).

   Ejemplo para cambiar el puerto a 9000 si es necesario:

   sonar.web.port=9000

Instalar SonarQube

- **Paso 3: Iniciar SonarQube**

1. Ve a la carpeta /bin y elige el script según tu SO, por ejemplo para Linux 64-bit:

```
cd /opt/sonarqube/bin/linux-x86-64/
```

2. Ejecuta el siguiente comando para iniciar SonarQube:

```
./sonar.sh start
```

3. Verifica el estado con:

```
./sonar.sh status
```

Instalar SonarQube

- **Paso 4: Acceder a SonarQube**
  - Abre tu navegador y entra a: http://localhost:9000
  - Usuario y contraseña por defecto son ambos: admin
  - Cambia la contraseña en el primer inicio para mayor seguridad.

- **Paso 5: (Opcional) Configurar base de datos externa**
  - Para producción, mejor usar bases de datos como PostgreSQL, en sonar.properties puedes configurar:

```
sonar.jdbc.username=usuario
sonar.jdbc.password=contraseña
sonar.jdbc.url=jdbc:postgresql://localhost/sonarqube
```

Icono *Training Consulting*

Testing Back

Completa nuestra encuesta de satisfacción a través del QR

GRACIAS

VIEWNEXT
AN IBM SUBSIDIARY

2025