



# Formación

## Curso Pruebas Unitarias



## 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](mailto:training@iconotc.com)

▣ **Duración:** 12 horas

▣ **Modalidad:** Presencial / Remoto

▣ **Fechas/Horario:**

- Días 16, 17 y 18 Febrero 2026
- Horario de 15:00 a 19:00 hs

▣ **Contenidos:**

- Introducción
- Junit
- Normativa de Test usada
- Mocks
  - Tipos de Mocks
  - Frameworks de Mocks
- Calidad de Test
  - Cobertura
  - Mutación Test
- Desarrollo Dirigido por Tests (TDD)
  - El Algoritmo TDD

# Introducción

Tema 1

## 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 Integración 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 Integración 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

## Tema 2

## 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

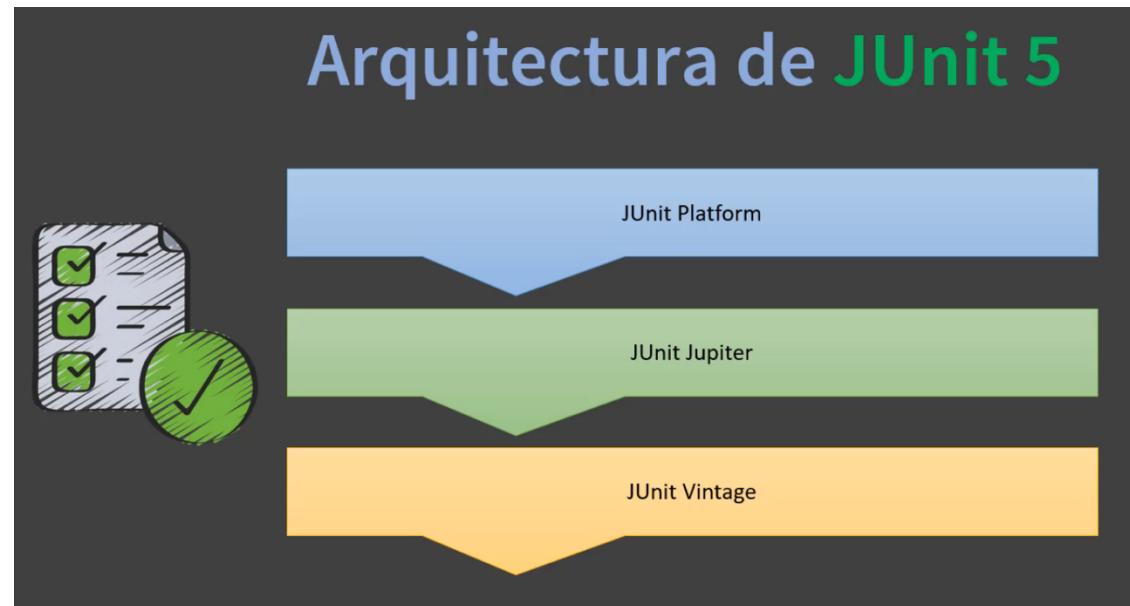
- Versión **JUnit4**:
  - Incluye anotaciones (Java 5 annotations) en lugar de utilizar herencia:
  - `@Test` sustituye a la herencia de `TestCase`.
  - `@Before` y `@After` como sustitutos de `setUp` y `tearDown`.
  - `@BeforeClass` y `@AfterClass`, metodos estáticos para inicializar y liberar recursos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

```
<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.

# **Normativa de Test usada**

Tema 3

## Normativa utilizada en Pruebas Unitarias

- La normativa utilizada en pruebas unitarias establece directrices para garantizar la calidad, consistencia y efectividad en el diseño y ejecución de estas pruebas. A continuación, se resumen los puntos clave basados en estándares y buenas prácticas:

## Principales Normas y Buenas Prácticas

### 1. Características de las pruebas unitarias:

- **Atómicas:** Cada prueba debe evaluar una funcionalidad mínima e independiente del sistema.
- **Independientes:** Las pruebas no deben depender unas de otras ni del orden en que se ejecutan.
- **Reproducibles:** Los resultados deben ser consistentes, independientemente del entorno o frecuencia de ejecución.
- **Rápidas:** Deben ejecutarse en poco tiempo para no afectar la productividad.

## Principales Normas y Buenas Prácticas

### 2. Enfoques de diseño:

- **Caja blanca:** Evalúa la estructura interna del código, como caminos lógicos o instrucciones no usadas.
- **Caja negra:** Valida entradas y salidas sin considerar la lógica interna del componente.

## Principales Normas y Buenas Prácticas

### 3. Estándares relevantes:

- **IEEE 1008:** Define un enfoque sistemático para las pruebas unitarias, incluyendo planificación, diseño, ejecución y evaluación de resultados. Este estándar asegura que todas las funcionalidades sean cubiertas por casos de prueba y especifica tareas como identificar riesgos, validar datos y medir cobertura.
- **ISO/IEC/IEEE 29119:** Proporciona un marco general para pruebas de software, incluyendo pruebas unitarias como parte integral del ciclo de vida del desarrollo.

## Principales Normas y Buenas Prácticas

### 4. Buenas prácticas adicionales:

- Verificar tanto condiciones válidas como inválidas.
- Evitar efectos secundarios que puedan alterar el estado del sistema.
- Mantener las pruebas legibles y fáciles de actualizar para adaptarse a cambios en el código.

# Mocks

Tema 4

## 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

```
<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>
</dependencies>
```

## 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");
```

# Calidad de Test

## Tema 5

## Métricas de test

- Los test son la única garantía que tengo de certificar que el código que he desarrollado funciona
- Las herramientas de métricas de test se basan en parámetros cualitativos y no en parámetros cuantitativos. Básicamente, la calidad de nuestros test está más relacionado con los casos de uso cubiertos que con las líneas cubiertas. El problema es que esto no es medible de manera sencilla.

## Cobertura de código

- La cobertura de código es un parámetro con el que podrás saber qué parte de tu fuente se ha sometido a pruebas. Es muy útil para evaluar la calidad del conjunto de pruebas.
- Este indicador mide el porcentaje de líneas que están cubiertas por test unitarios en nuestro código. Generalmente, se asume que un porcentaje alto de este indicador se traduce en calidad.
- La cobertura de código indica lo minucioso que se ha sido para cubrir el máximo número de líneas pero no indica que las pruebas que se estén haciendo sirva para algo.
- En resumen, la cobertura de código en sí misma no es un indicador de calidad.

## Como se calcula la Cobertura de código

- Las herramientas de cobertura de código utilizarán uno o varios criterios para determinar cómo se ejecutó o no tu código durante la ejecución del conjunto de pruebas. Estos son algunos de los parámetros que es habitual encontrar en los informes de cobertura:
  - **Cobertura de funciones:** la cantidad de funciones definidas que se abarcan.
  - **Cobertura de instrucciones:** cuántas instrucciones del programa se han ejecutado.
  - **Cobertura de ramas:** cuántas ramas de las estructuras de control (instrucciones IF, por ejemplo) se han ejecutado.
  - **Cobertura de condiciones:** en cuántas subexpresiones booleanas se ha comprobado el valor verdadero o falso.
  - **Cobertura de líneas:** cuántas líneas de código fuente se han probado.

## Porcentaje de Cobertura de código

- Generalmente se acepta que la cobertura del 80 % es el objetivo que se debe perseguir. Tratar de alcanzar una cobertura mayor puede resultar costoso y no siempre genera unas ventajas que justifiquen el esfuerzo.

## Herramientas para Cobertura de código

- Java: Atlassian Clover, Cobertura, JaCoCo
- JavaScript: istanbul
- PHP: PHPUnit
- Python: Coverage.py
- Ruby: SimpleCov

## Mutation test

- Esta técnica consiste en introducir modificaciones en el código fuente de los test unitarios y verificar que fallan al ejecutarlos con el código mutado.
- Las mutaciones introducidas comúnmente son:
  - Reemplazo de valores lógicos: True por False por ejemplo.
  - Reemplazo de operadores aritméticos.
  - Reemplazo de condiciones lógicas (>,<,etc.)
  - Eliminar líneas de código.
  - Alterar contadores.
- Esto evita principalmente que nuestros test sean muy vagos y que se construyan únicamente con la intención de llegar a un determinado valor de cobertura de código.

# Desarrollo dirigido por Tests (TDD)

Tema 6

## 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.

# Pruebas Unitarias



Completa nuestra encuesta  
de satisfacción a través del QR

Curso Pruebas Unitarias (16-18 feb)



GRACIAS

**VIEWNEXT**  
AN IBM SUBSIDIARY