



# PRUEBAS UNITARIAS

LUED BELTRAN  
ROGER ESCOBAR



Las pruebas unitarias son un tipo de prueba de software en la que se verifica el funcionamiento de componentes individuales o módulos del código de forma aislada. Su propósito es asegurarse de que cada parte del sistema funcione correctamente antes de integrarse con otros módulos.

## ¿QUÉ SON LAS PRUEBAS UNITARIAS?

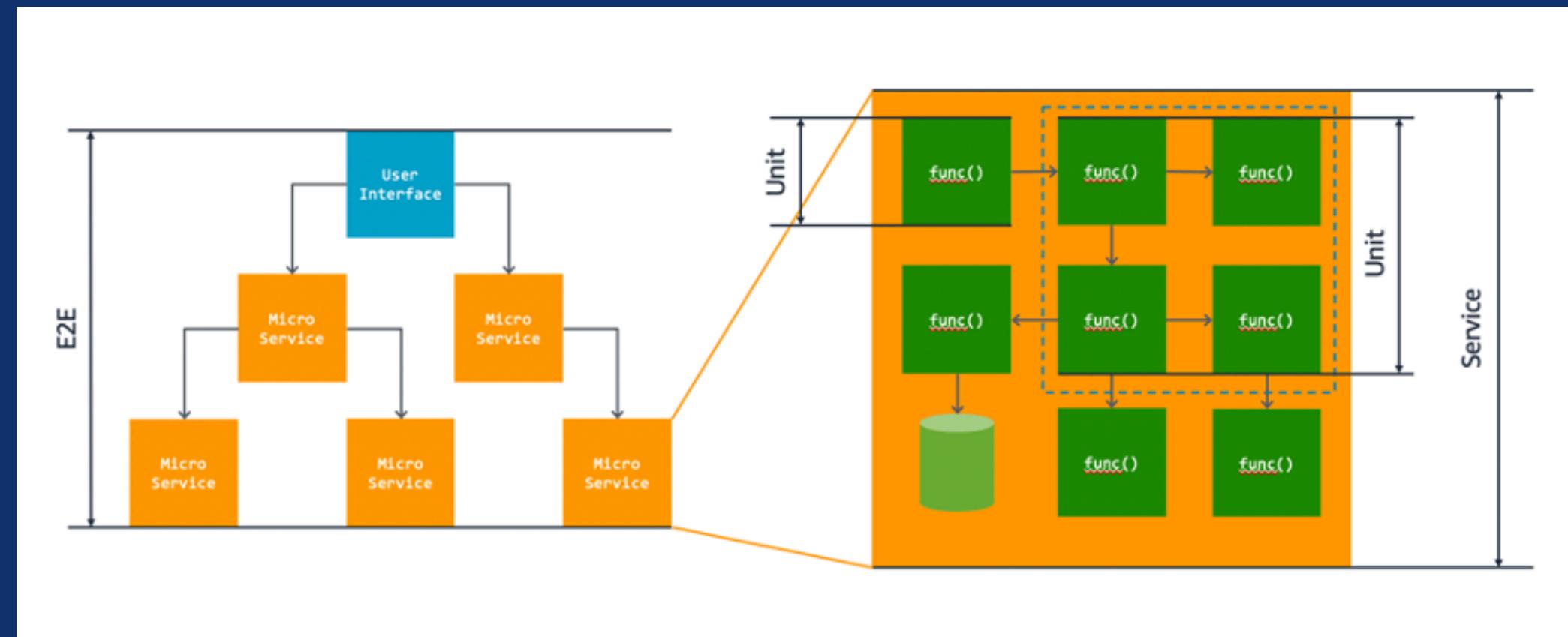
**son esenciales porque:**

- Detectan errores en etapas tempranas del desarrollo, reduciendo costos de corrección.
- Facilitan el mantenimiento y la refactorización del código sin afectar funcionalidades previas.
- Mejoran la calidad del software y la confianza en el código.





# UNA PRUEBA UNITARIA ES UN BLOQUE DE CÓDIGO QUE VERIFICA LA PRECISIÓN DE UN BLOQUE MÁS PEQUEÑO Y AISLADO DE CÓDIGO DE APLICACIÓN, NORMALMENTE UNA FUNCIÓN O UN MÉTODO.



La prueba unitaria está diseñada para verificar que el bloque de código se ejecuta según lo esperado, de acuerdo con la lógica teórica del desarrollador. La prueba unitaria solo interactúa con el bloque de código a través de entradas y salidas (verdaderas o falsas) capturadas afirmadas.

Un solo bloque de código también puede tener un conjunto de pruebas unitarias, conocidas como casos de prueba. Un conjunto completo de casos de prueba cubre todo el comportamiento esperado del bloque de código, pero no siempre es necesario definir el conjunto completo de casos de prueba.



# PRINCIPALES PRUEBAS UNITARIAS, APLICACIÓN Y CARACTERÍSTICAS

## PRUEBAS DE CAJA BLANCA:

- Analizan el código fuente y su flujo de ejecución.
- Se enfocan en estructuras internas, condiciones, bucles y excepciones.
- Ejemplo: Verificar que una función de cálculo de impuestos devuelve el valor esperado para diferentes tasas impositivas.

## PRUEBAS DE CAJA NEGRA:

- Evalúan la funcionalidad sin conocer el código interno.
- Se basan en entradas y salidas esperadas.
- Ejemplo: Probar una función que convierte monedas asegurándose de que los valores sean correctos.





# PRINCIPALES PRUEBAS UNITARIAS, APLICACIÓN Y CARACTERÍSTICAS

## PRUEBAS DE MUTACIÓN:

- Introducen modificaciones en el código para evaluar la efectividad de las pruebas existentes.
- Si las pruebas no detectan el cambio, significa que deben mejorarse.
- Ejemplo: Cambiar una condición lógica en un if y verificar si las pruebas fallan.

## PRUEBAS PARAMÉTRICAS:

- Permiten ejecutar la misma prueba con diferentes conjuntos de datos.
- Mejoran la cobertura de pruebas y detectan casos límite.
- Ejemplo: Probar una función de división con diferentes numeradores y denominadores.



# ¿CÓMO CREAR PRUEBAS UNITARIAS?

Para crear pruebas unitarias, puede seguir algunas técnicas básicas para garantizar que se consideren todos los posibles escenarios de casos de prueba.

## Verificaciones Lógicas

- ¿El sistema realiza los cálculos correctos y sigue la ruta adecuada en el código cuando se le proporciona una entrada correcta y esperada?
- ¿Las entradas proporcionadas consideran todas las posibles rutas del código?

## Verificación de los límites

- ¿Cómo responde el sistema a las entradas proporcionadas?
- ¿Cómo responde a las entradas típicas, los casos periféricos o las entradas inválidas?
- ¿Cómo responde el sistema a las entradas proporcionadas?
- ¿Cómo responde a las entradas típicas, los casos periféricos o las entradas inválidas?
- ¿Cómo responde el sistema a las entradas proporcionadas?
- ¿Cómo responde a las entradas típicas, los casos periféricos o las entradas inválidas?

## Manejo de errores

- ¿Cómo responde el sistema cuando hay errores en las entradas?
- ¿Se le pide al usuario que introduzca otra entrada?
- ¿Se bloquea el software?

## Verificaciones orientadas a objetos

Si el estado de algún objeto persistente cambia al ejecutar el código,

- ¿se actualiza el objeto como corresponde?



# EJEMPLO DE PRUEBA UNITARIA

- EJEMPLO: FUNCIÓN PARA VERIFICAR SI UN NÚMERO ES PAR.  
ESTA FUNCIÓN TOMA UN NÚMERO Y DEVUELVE TRUE SI ES PAR Y FALSE SI ES IMPAR.

```
import pytest

# Método a probar
def is_even(number):
    return number % 2 == 0

# Pruebas unitarias
def test_even_number():
    assert is_even(4) == True

def test_odd_number():
    assert is_even(7) == False

def test_zero():
    assert is_even(0) == True # Cero es
un número par
```

```
def test_negative_even():
    assert is_even(-10) == True

def test_negative_odd():
    assert is_even(-3) == False

# Alternativa usando pytest.mark.parametrize
para simplificar
@pytest.mark.parametrize("number, expected", [
    (4, True),
    (7, False),
    (0, True),
    (-10, True),
    (-3, False)
])
def test_is_even(number, expected):
    assert is_even(number) == expected
```

# ¿CUÁLES SON LAS VENTAJAS ... DE LAS PRUEBAS UNITARIAS?

## DETECCIÓN EFICAZ DE ERRORES

Las pruebas unitarias detectan errores en el código antes de que lleguen a producción, permitiendo identificar problemas de entrada, salida o lógica. Cuando el código cambia, estas pruebas se ejecutan nuevamente junto con otras, como las de integración, asegurando que los resultados sean consistentes. Si fallan, indican errores por regresión. Además, facilitan la detección rápida de fallos, reduciendo el tiempo de depuración y ayudando a los desarrolladores a encontrar con precisión la causa del problema.



# ¿CUÁLES SON LAS VENTAJAS DE LAS PRUEBAS UNITARIAS?



## DOCUMENTACIÓN

Es importante documentar el código para saber exactamente lo que se supone que debe hacer ese código. Dicho esto, las pruebas unitarias también actúan como una forma de documentación.

Otros desarrolladores leen las pruebas para ver qué comportamientos se espera que muestre el código cuando se ejecute.

Utilizan la información para modificar o refactorizar el código. Refactorizar el código hace que sea más eficaz y esté mejor compuesto. Puede volver a ejecutar las pruebas unitarias para verificar que el código funciona según lo esperado después de los cambios.



# ¿CÓMO UTILIZAR LAS PRUEBAS UNITARIAS?



Las pruebas unitarias se implementan mediante la escritura de casos de prueba automatizados que validan el correcto funcionamiento de funciones o módulos específicos de un software en aislamiento. Se utilizan frameworks como JUnit para Java, pytest para Python o NUnit para .NET, permitiendo definir entradas controladas y comparar los resultados con valores esperados. Los desarrolladores ejecutan estas pruebas de forma recurrente, especialmente después de realizar cambios en el código, asegurando que cada componente sigue cumpliendo su propósito sin introducir fallos. En entornos de integración continua, las pruebas unitarias se integran en pipelines de desarrollo, detectando errores de manera temprana y evitando regresiones antes del despliegue en producción.

# LOS DESARROLLADORES UTILIZAN LAS PRUEBAS UNITARIAS EN VARIAS ETAPAS DEL CICLO DE VIDA DEL DESARROLLO DE SOFTWARE.

## DESARROLLO BASADO EN PRUEBAS

01

El desarrollo basado en pruebas (TDD) consiste en que los desarrolladores crean pruebas para verificar los requisitos funcionales de un programa antes de crear el código completo. Al escribir primero las pruebas, el código se puede verificar al instante en función de los requisitos, una vez que se realiza la codificación y se ejecutan las pruebas.

## DESPUÉS DE COMPLETAR UN BLOQUE DE CÓDIGO

02

Una vez que un bloque de código se considera completo, deben llevarse a cabo pruebas unitarias, si es que aún no se han hecho, mediante el TDD. Luego, puede ejecutar pruebas unitarias al instante para verificar los resultados. Las pruebas unitarias también se ejecutan como parte del conjunto completo de otras pruebas de software durante las pruebas del sistema. Por lo general, son el primer conjunto de pruebas que se ejecutan durante las pruebas de software del sistema completo.

# LOS DESARROLLADORES UTILIZAN LAS PRUEBAS UNITARIAS EN VARIAS ETAPAS DEL CICLO DE VIDA DEL DESARROLLO DE SOFTWARE.

## EFICIENCIA DE DEVOPS

03

Las pruebas unitarias son un componente esencial dentro del conjunto de pruebas de software, trabajando en conjunto con las pruebas de integración para garantizar la estabilidad del sistema. En entornos de CI/CD (Integración y Entrega Continua), estas pruebas se ejecutan de forma automatizada cada vez que se actualiza el código, detectando errores de manera temprana y asegurando su calidad a lo largo del tiempo. Esto permite a los desarrolladores implementar cambios con confianza, minimizando riesgos y evitando regresiones antes del despliegue en producción.



# ¿CUÁLES SON LAS PRACTICAS RECOMENDADAS DE PRUEBAS UNITARIAS?



## 10 MEJORES PRÁCTICAS

## PRUEBAS INDEPENDIENTES Y AISLADAS

01

Cada prueba debe evaluar un único componente sin depender de otros módulos o estados compartidos. Esto garantiza que los resultados sean consistentes y que un fallo en una prueba no afecte a otras. El aislamiento permite una ejecución más rápida y facilita la depuración en caso de errores.

## COBERTURA DE CÓDIGO ADECUADA

Si bien una alta cobertura de código es deseable, no se debe priorizar la cantidad sobre la calidad. Es importante enfocarse en las funciones críticas del sistema y en casos límite, asegurando que los componentes más relevantes estén bien probados sin generar pruebas redundantes o innecesarias.

02



03

## USO DE DATOS CONTROLADOS

Las pruebas unitarias deben ejecutarse con datos conocidos y predecibles, evitando el uso de valores aleatorios o dependencias externas. Esto asegura consistencia en los resultados y facilita la detección de errores al eliminar incertidumbres en la entrada.



04

## AUTOMATIZACIÓN Y EJECUCIÓN FRECUENTE

Integrar las pruebas unitarias en la canalización de CI/CD permite ejecutarlas de manera automática cada vez que se actualiza el código. Esto ayuda a detectar errores de forma temprana y evita que cambios recientes generen problemas en funcionalidades ya validadas.

05

## NOMBRAZO CLARO Y DESCRIPTIVO

Los nombres de las pruebas deben ser explícitos y reflejar exactamente qué se está validando. Un buen nombre, como `test_calculate_discount_when_user_is_premium()`, facilita la comprensión y mantenimiento del código sin necesidad de revisar su contenido.

06

## ESTRUCTURA AAA (ARRANGE-ACT-ASSERT)

Este patrón mejora la claridad y organización de las pruebas. Primero, se configuran los datos de entrada (Arrange), luego se ejecuta la funcionalidad a probar (Act) y finalmente se verifican los resultados esperados (Assert). Seguir este esquema evita ambigüedades y facilita la lectura del código de prueba.

07

## EVITAR LÓGICA COMPLEJA EN LAS PRUEBAS

Las pruebas unitarias deben ser lo más simples posible, evitando condicionales, bucles u operaciones innecesarias. Una prueba demasiado compleja puede contener errores propios, dificultando la depuración y reduciendo su fiabilidad como mecanismo de validación.

08

## MANEJO DE DEPENDENCIAS CON MOCKS Y STUBS

Para probar módulos que dependen de bases de datos, APIs o servicios externos, es recomendable usar Mocks o Stubs. Estas herramientas simulan respuestas controladas, evitando dependencias innecesarias y asegurando que las pruebas sean rápidas y confiables.

09

## EJECUCIÓN RÁPIDA Y EFICIENTE

Las pruebas unitarias deben ejecutarse en milisegundos. Si una prueba tarda demasiado, puede estar incluyendo lógica de integración, lo que va en contra de su propósito. Mantenerlas ligeras y optimizadas contribuye a un desarrollo ágil y continuo.

10

## MANTENER Y ACTUALIZAR LAS PRUEBAS

Las pruebas deben evolucionar junto con el código. Una prueba que ya no es relevante debe ser actualizada o eliminada para evitar falsos positivos o negativos. Mantener un conjunto de pruebas actualizado y relevante garantiza su efectividad en la detección de errores. Siguiendo estas prácticas, las pruebas unitarias se convierten en una herramienta clave para asegurar la calidad y estabilidad del software a lo largo del tiempo.

# ¿CUÁL ES LA DIFERENCIA ENTRE LAS PRUEBAS UNITARIAS Y LOS OTROS TIPOS DE PRUEBAS?

Existen muchos otros métodos de prueba de software además de las pruebas unitarias. Todos desempeñan funciones específicas en el ciclo de vida del desarrollo de software:



- Las pruebas de integración verifican que las diferentes partes del sistema de software que están diseñadas para interactuar, lo hagan correctamente.
- Las pruebas funcionales verifican que el sistema de software cumpla con los requisitos de software descritos antes de su creación.
- Las pruebas de rendimiento verifican que el software funcione según los requisitos de rendimiento esperados, como la velocidad y el tamaño de la memoria.
- Las pruebas de aceptación se producen cuando las partes interesadas o los grupos de usuarios prueban manualmente el software para comprobar que funcione según lo previsto.
- Las pruebas de seguridad verifican el software para detectar vulnerabilidades y amenazas conocidas. Esto incluye el análisis de la superficie de amenazas, incluidos los puntos de entrada de terceros al software.



**GRACIAS**  
**POR SU ATENCIÓN**