Engineer

Prueba unitaria

Prueba automática de una pequeña unidad de código aislada

Qué es

Una prueba unitaria es una pequeña comprobación automática de una parte del código por separado: llama a una función, una clase o un módulo pequeño con entradas elegidas y comprueba mediante aserciones que lo devuelto sea lo esperado. Estas pruebas suelen diseñarse para ejecutarse en el mismo proceso que el código comprobado, terminar deprisa y no necesitar red, base de datos ni navegador. Son decisiones útiles sobre recursos y aislamiento, no garantías que se deriven únicamente del alcance reducido de una prueba. La práctica de escribirlas suele llamarse pruebas unitarias y la colección de un proyecto constituye su batería de pruebas unitarias.

Qué se considera una unidad está menos claro de lo que sugiere el nombre. En el capítulo sobre pruebas unitarias de *Software Engineering at Google*, Erik Kuefler emplea el término para pruebas de alcance relativamente reducido, como una sola clase o método, y añade que una unidad puede ser una función o abarcar varios paquetes relacionados. Para delimitar una unidad, resulta útil fijarse en lo que invocan sus consumidores: una unidad es lo que ofrece una interfaz de la que depende alguien externo a ella, y sus pruebas utilizan esa interfaz como lo hacen sus consumidores.

Los frameworks modernos comparten un origen. El artículo de Kent Beck *Simple Smalltalk Testing: With Patterns* describía un pequeño framework formado por preparaciones de prueba, casos de prueba, comprobaciones y conjuntos de pruebas, y aconsejaba que los desarrolladores escribieran sus propias pruebas unitarias, una por clase. La documentación del `unittest` de Python indica que se inspiró originalmente en JUnit y conserva las mismas cuatro ideas: una preparación de prueba o fixture —los preparativos que necesita una prueba y la limpieza posterior—, un caso de prueba —la unidad individual de prueba—, un conjunto de pruebas y un ejecutor. Ambos distinguen algo que conviene

mantener: un fallo es una comprobación que obtiene una respuesta incorrecta; un error es algo que la prueba no anticipaba, como una excepción lanzada desde lo más profundo del código.

Las pruebas unitarias ocupan el extremo de menor alcance de una escala. El capítulo sobre pruebas de Adam Bender en el mismo libro las clasifica por alcance: las de alcance reducido suelen llamarse unitarias; las de alcance intermedio, que cubren varios componentes juntos, son pruebas de integración; y las de alcance amplio, que manejan el sistema completo, son pruebas de extremo a extremo.

Utilice el concepto en una decisión

Aplique «¿Qué es una prueba unitaria?» a una decisión real. Podemos ayudarle a relacionar el concepto con el sistema que está construyendo o comprando. Un equipo que lleva una idea del boceto a un producto al que sus clientes pueden acceder, sin pasarla entre tres proveedores.

Explore nuestros servicios Póngase en contacto →

Compruebe los supuestos detrás del término

Utilice la explicación y los trabajos enlazados para formular preguntas sobre su proyecto. Podemos examinar los supuestos con usted antes de elegir una herramienta. Ingresos multiplicados por 10 con mapas GIS — 10x crecimiento de ingresos impulsado desde el producto cartográfico

Ingresos multiplicados por 10 con mapas GIS Póngase en contacto →

Por qué importa

La rapidez es la primera razón. Una prueba unitaria pequeña y aislada se ejecuta en el mismo proceso que el código comprobado, de modo que un ingeniero puede ejecutar miles durante una jornada normal y detectar un error mientras el cambio que lo causó sigue en pantalla. Las directrices de Google son tajantes sobre lo permitido: una prueba pequeña no debe hacer pausas, realizar E/S ni acceder a la red o al disco. Estas restricciones facilitan construir pruebas rápidas y repetibles, y un equipo aprende a confiar en una batería que da siempre la misma respuesta.

La segunda razón es el diagnóstico. Cuando falla una prueba de alcance reducido, el defecto está en una parte pequeña del código, normalmente la que acaba de cambiar. Cuando falla una prueba que maneja el sistema completo, el defecto puede estar en cualquier lugar entre el navegador y la base de datos. Por eso, *Software Engineering at Google* propone una pirámide como orientación aproximada: en torno al 80 % de pruebas de alcance reducido, un 15 % de integración y un 5 % de extremo a extremo.

La tercera es el cambio. Una batería que comprueba el comportamiento a través de una interfaz pública permite reestructurar el código que hay detrás con confianza: si las pruebas siguen pasando, los consumidores siguen atendidos. El ideal de Kuefler es una prueba que nunca deba cambiar salvo que cambien los requisitos, de modo que una refactorización pura no toque ninguna prueba.

Las pruebas unitarias también documentan. Una prueba cuyo nombre describe un comportamiento, con entradas claras y un resultado esperado escrito por completo, muestra a quien viene después cómo debe llamarse al código y qué promete; a diferencia de un comentario, falla cuando deja de ser cierto.

Cómo se aplica en la práctica

Considere una función que lee duraciones como `1h30m` de un archivo de configuración y las convierte en segundos:

```
# durations.py
UNITS = {"h": 3600, "m": 60, "s": 1}


def to_seconds(text: str) -> int:
    """Turn "1h30m" or "45s" into a whole number of seconds."""
    total, digits = 0, ""
    for char in text:
        if char.isdigit():
            digits += char
        elif char in UNITS and digits:
            total += int(digits) * UNITS[char]
            digits = ""
        else:
            raise ValueError(f"not a duration: {text!r}")
    if digits or not text:
        raise ValueError(f"not a duration: {text!r}")
    return total
```

Sus pruebas unitarias, escritas para pytest, comprueban dos comportamientos: qué acepta y qué rechaza.

```
# test_durations.py
import pytest

from durations import to_seconds
```

```
@pytest.mark.parametrize(
    "text,expected",
    [("45s", 45), ("2m", 120), ("1h30m", 5400), ("0s", 0)],
)
def test_counts_each_unit(text, expected):
    assert to_seconds(text) == expected


@pytest.mark.parametrize("text", ["", "90", "h", "1x", "1.5h"])
def test_refuses_what_it_cannot_read(text):
    with pytest.raises(ValueError):
        to_seconds(text)
```

Ejecutar `pytest` en el directorio recoge todos los archivos llamados `test_*.py` o `*_test.py` ; las dos funciones se expanden en nueve casos, uno por cada fila de parámetros. La documentación de pytest describe cada prueba mediante preparación, acción, aserción y limpieza. Aquí, la preparación es la fila de parámetros, la acción es una única llamada y no hay nada que limpiar porque no se toca nada fuera de la función. Si alguien cambia por error el minuto a 3.600 segundos, fallan dos casos y el informe identifica cada uno mediante sus parámetros: `test_counts_each_unit[2m-120]` muestra 7.200 frente a los 120 esperados y `test_counts_each_unit[1h30m-5400]` muestra 111.600 frente a 5.400. Observe que los valores esperados se calculan a mano, no se copian de lo que devolvió el código al ejecutarse por primera vez.

Las mismas comprobaciones se implementarían con el `unittest` de la biblioteca estándar en una subclase de `unittest.TestCase` , con métodos cuyos nombres empiezan por `test` y que utilizan `assertEqual` y `assertRaises` ; `python -m unittest` encuentra de forma predeterminada los archivos que coinciden con `test*.py` . Go incorpora la convención a sus herramientas: una función `func TestXxx(t *testing.T)` situada en un archivo terminado en `_test.go` se ejecuta mediante `go test` .

La escala cambia la organización, no la idea. El generador de base de datos sujeto a 981 casos de prueba, un trabajo de julio de 2026, cuenta con 383 funciones de prueba que se convierten en 981 casos al expandirse las parametrizadas, distribuidas en 45 archivos y 7.944 líneas. Su compilación exige un mínimo de cobertura de ramas del 92 %, con alrededor del 95 % medido, y trata las advertencias como fallos. La página expone claramente el límite de la cifra: dos defectos encontrados más tarde estaban en código que el informe de cobertura mostraba como cubierto.

Dónde surgen los problemas

Las pruebas reproducen la implementación. Una prueba que entra en funciones auxiliares privadas o exige la secuencia exacta de llamadas internas se rompe cada vez que se reestructura el código, aunque no cambie nada visible para un consumidor. Kuefler llama a esto una prueba frágil y señala la dependencia excesiva de frameworks de simulación como la causa más habitual de los problemas de las pruebas de interacciones. Pruebe a través de la interfaz pública, compruebe el estado resultante en lugar de las llamadas utilizadas para obtenerlo y use objetos reales cuando su coste sea bajo.

Las pruebas dependen unas de otras. Una crea un registro, la siguiente lo lee y la batería solo pasa en ese orden. Al ejecutar una prueba sola o las mismas pruebas en otro orden, falla sin una razón que quien lee pueda ver. El framework de Beck creaba y descartaba los datos de prueba para cada prueba, y la documentación de `unittest` exige que un caso sea completamente autónomo. Dé a cada prueba su propia preparación y limpie después.

La cobertura se convierte en el objetivo. La cobertura muestra que una línea se ejecutó, no que alguien comprobara lo que hizo. Una prueba sin aserciones aumenta la cifra y no demuestra nada; *Software Engineering at Google* advierte de que un umbral del 80 % tiende a convertirse en un techo, con cambios que alcanzan exactamente el mínimo. Trate la cifra como un suelo y lea las aserciones durante la revisión.

Una prueba por método. Emparejar cada método de producción con un método de prueba parece ordenado, pero la prueba crece cada vez que el método añade una rama, hasta que nadie sabe decir qué comprueba. Kuefler aconseja probar comportamientos: uno por prueba, con un nombre que indique lo que demuestra, para que el informe de fallos se lea como una oración sobre lo que se rompió.

La prueba espera a que pase el tiempo. Un `sleep()` para dejar que algo termine ralentiza la batería en cada ejecución y sigue fallando en una máquina ocupada. En una prueba unitaria, el tiempo es una entrada: pase un reloj o una marca de tiempo para que la prueba fije el instante que necesita en lugar de esperarlo.

Lecturas adicionales

- unittest: framework de pruebas unitarias — el framework de la biblioteca estándar de Python: preparaciones, casos, conjuntos, ejecutores, descubrimiento y diferencia entre fallo y error.
- Anatomía de una prueba — la explicación de pytest sobre preparación, acción, aserción y limpieza.
- Primeros pasos — la introducción de pytest: `assert` sin añadidos, descubrimiento de pruebas y `pytest.raises`.
- Pruebas unitarias — el capítulo de Erik Kuefler en *Software Engineering at Google* sobre pruebas frágiles, API públicas, estado frente a interacciones y pruebas de comportamientos.
- Simple Smalltalk Testing: With Patterns — el artículo de Kent Beck sobre su framework, las razones para escribir una prueba unitaria por clase y la distinción entre fallos y errores, en una copia archivada.
- testing — el paquete de pruebas estándar de Go: funciones `TestXxx`, archivos `_test.go` y `go test`.

Glosario

Elija según sus restricciones reales

Describa el flujo de trabajo, las interfaces existentes y los usuarios que necesitan el cambio. Podemos definir una entrega comprobable y un traspaso que permita a su equipo mantener lo entregado.

Vea cuánto cuesta Póngase en contacto →

Haga práctica la terminología

Pregunte cómo se aplica «¿Qué es una prueba unitaria?» a su organización. Basta un ejemplo concreto de su sistema actual para iniciar la conversación.

Explore nuestros servicios

welcome@engineer.company +45 50 10 61 84 Copenhague, Dinamarca

Llévese esta página

Tarjeta de contacto

Creado con esmero y cariño por D.

CVR 45969703 · Tagensvej 105C, 3. th, 2200 København N

Derechos de autor © 2025 – actualidad · Engineer ApS · Todos los derechos reservados.

Impreso desde https://engineer.company/es/glossary/unit-test/