



Tests Unitarios

**El Arte de Escribir Tests Unitarios: Tu
guía para un Código a Prueba de
Errores**




PABLO DEL ÁLAMO



Introducción

¿Qué Son los Tests Unitarios?

Imagínate a cada parte de tu código como una pieza de rompecabezas. Un test unitario asegura que cada pieza encaje perfectamente antes de unirlos.

Comprueban el funcionamiento de las funciones individuales para que todo vaya sobre ruedas. 



PABLO DEL ÁLAMO



¿Por Qué Usar Tests Unitarios?

Son un remedio infalible contra errores silenciosos.

Se anticipan a problemas futuros, documentan el comportamiento esperado y reducen el esfuerzo al refactorizar.



PABLO DEL ÁLAMO



Ventajas del Testing Unitario

Hace, que el código sea más robusto, transparente y libre de sorpresas. Evitan regresiones, mejoran la calidad del software y ahorran tiempo en el largo plazo. En resumen, son un salvavidas. 🚢



PABLO DEL ÁLAMO



¿Cuándo Hacer Tests Unitarios?

En un mundo ideal, junto con el código nuevo (siguiendo o no TDD), pero nunca es tarde para implementarlos.

Incluso en proyectos antiguos, pueden ser grandes aliados para evitar romper algo sin querer, al introducir o alterar funcionalidades.



PABLO DEL ÁLAMO



Herramientas para Implementarlos

Es cuestión de gusto: si estás en JavaScript, echa un vistazo a Jest o Mocha; para Java, dale una oportunidad a JUnit; en Python, tienes unittest y pytest. La elección es tuya.



PABLO DEL ÁLAMO



Ejemplo en Python (Parte 1)

Démosle vida a una función: `sumar(a, b)` que suma dos números. ¿Cómo lo testamos?

Código:

```
def suma(a, b):  
    return a + b
```



PABLO DEL ÁLAMO



Ejemplo en Python (Parte 2)

Creamos el test unitario:

```
import unittest

class TestSuma(unittest.TestCase):
    def test_suma(self):
        self.assertEqual(suma(2, 3), 5)
```



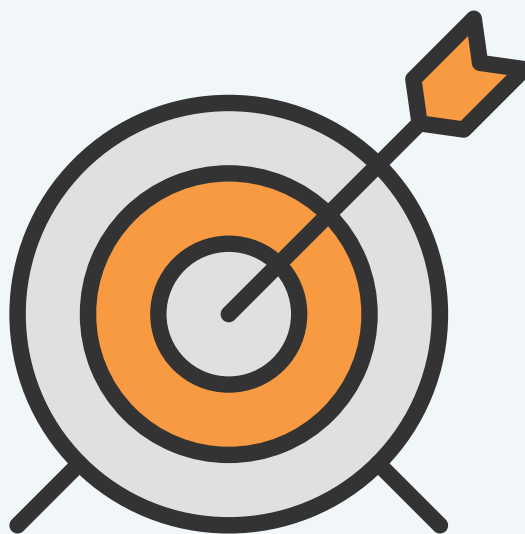
PABLO DEL ÁLAMO



Buenas Prácticas al Hacer Tests Unitarios

Cada test debe ser autosuficiente y limpio, sin dependencias externas.

Mantén cada test claro, arguméntalo solo cuando sea necesario y asegúrate de que cubra un único caso de uso.



PABLO DEL ÁLAMO



Consejos para Principiantes

Empieza probando funciones básicas y de ahí expande tus tests a casos más complejos.

Esto no solo fortalece tu código, sino que también incrementa tu confianza. Da pasos firmes, no haya prisa.



PABLO DEL ÁLAMO



Cómo Escribir un Buen Test

Escribe tests que sean claros y tengan un solo propósito para fallar.

Un buen test es también parte de la documentación de tu código. Tenlo como una referencia para tu futuro yo 😊



PABLO DEL ÁLAMO



La Importancia de los Mocks y Stubs

Cuando testes funcionalidades que dependen de sistemas externos, utiliza mocks o stubs.

Simulan estos sistemas para que solo te centres en la funcionalidad interna.



PABLO DEL ÁLAMO



Ejemplo de Mock en Python (Parte 1)

Supongamos que tienes una función que obtiene datos de una API externa.

Queremos probar esta función sin depender de la API real, por lo que usaremos un mock.



PABLO DEL ÁLAMO



```
# Código a probar
import requests

def get_user_data(user_id):
    """Función que obtiene datos de un usuario desde una API externa."""
    response = requests.get(f"https://api.example.com/users/{user_id}")
    if response.status_code == 200:
        return response.json()
    else:
        return None
```

Aquí simularemos la respuesta de `requests.get` para que no haga una llamada real a la API



PABLO DEL ÁLAMO

```
class TestGetUserData(unittest.TestCase):
    @patch('requests.get') # Mockeamos la función requests.get
    def test_get_user_data_success(self, mock_get):
        # Configuramos el mock para devolver una respuesta simulada
        mock_get.return_value.status_code = 200
        mock_get.return_value.json.return_value = {"id": 1, "name": "John Doe"}

        # Llamamos a la función que estamos probando
        result = get_user_data(1)

        # Validamos que el resultado es el esperado
        self.assertEqual(result, {"id": 1, "name": "John Doe"})
        mock_get.assert_called_once_with("https://api.example.com/users/1")

    @patch('requests.get')
    def test_get_user_data_failure(self, mock_get):
        # Simulamos un error en la API
        mock_get.return_value.status_code = 404

        # Llamamos a la función
        result = get_user_data(1)

        # Validamos que el resultado sea None
        self.assertIsNone(result)
        mock_get.assert_called_once_with("https://api.example.com/users/1")

if __name__ == '__main__':
    unittest.main()
```



Coverage en Tests Unitarios

La cobertura (coverage) indica qué porcentaje de tu código está cubierto por los tests unitarios.

¿Cobertura del 100%? Sería lo ideal, pero no te obsesiones. Es preferible calidad sobre cantidad.

Intenta acercarte lo más posible (más de un 80% ya es un buen número), cubre casos críticos y asegúrate de que los tests sean útiles, no solo números en un reporte.

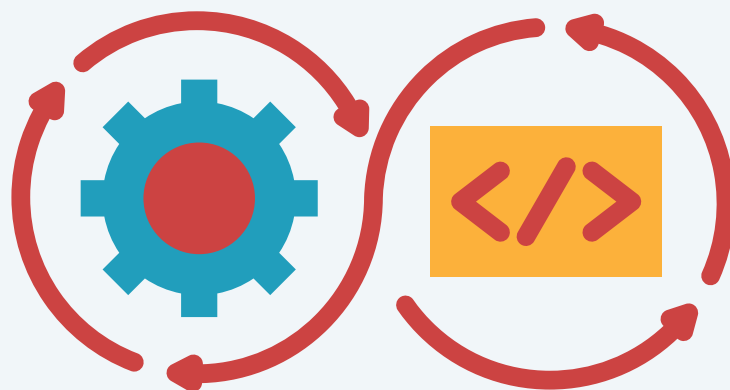


PABLO DEL ÁLAMO



Automatización de Tests (Parte 1)

Incorpora tus tests en pipelines CI/CD para que sean automáticos. Jenkins, Travis CI y GitHub Actions están ahí para hacer ese trabajo por ti mientras te tomas un café. (Si te gustaría que haga un post hablando en detalle sobre esto, házmelo saber 😊)

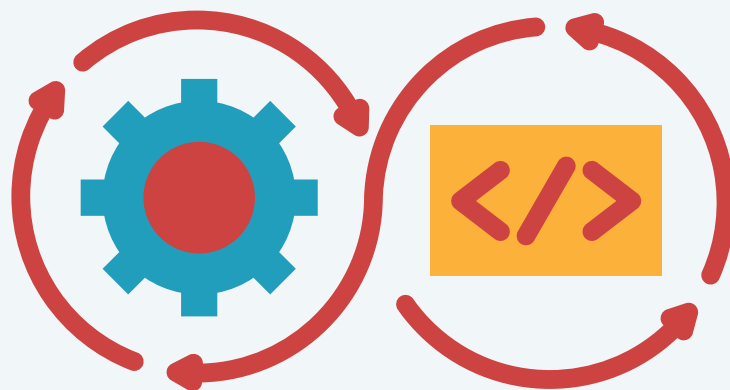


PABLO DEL ÁLAMO



Automatización de Tests (Parte 2)

Los tests en CI/CD no solo garantizan que tu código funcione tras cada cambio, sino que también detectan problemas de integración a tiempo.



PABLO DEL ÁLAMO

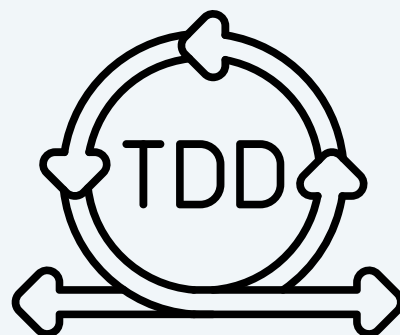


Prácticas TDD (Parte 1)

Con la metodología de Test Driven Development, escribes tests antes del código productivo.

Esto ayuda a clarificar requisitos y a garantizar que el código hace exactamente lo que debe.

Tengo un post hablando en detalle sobre TDD en mi perfil, por si te interesa 😊



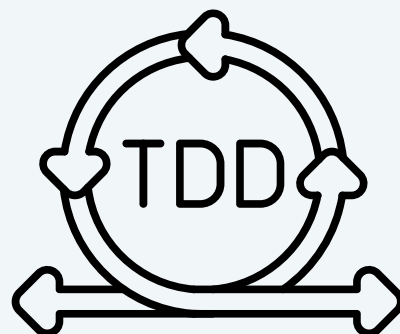
PABLO DEL ÁLAMO



Prácticas TDD (Parte 2)

El flujo es: crear un test que falle, escribir el código mínimo para pasarlo y luego refactorizar.

Este ciclo asegura que cada nueva funcionalidad esté bien definida desde el principio. 🔄



PABLO DEL ÁLAMO



Cuándo No Hacer Tests Unitarios

Si la lógica es extremadamente simple o completamente dependiente de servicios externos, podrías invertir tu esfuerzo en una capa más alta de testing, como pruebas de integración.

Evita el overtesting.



PABLO DEL ÁLAMO

Conclusión



En resumen, los tests unitarios son uno de tus mejores aliados en el mundo del desarrollo.

No solo te ofrecen la confianza de que tu código funciona como debería, sino que también facilitan el mantenimiento y la expansión del mismo.

Al incorporar estos tests, no solo mejoras la calidad de tu software, sino que también optimizas tu flujo de trabajo y aseguras una entrega más segura.

No olvides que, al final del día, el objetivo es entregar un producto en el que puedas confiar y que tus usuarios disfruten plenamente 🚀💻👥



PABLO DEL ÁLAMO



¿Te ha resultado útil?



- Comparte esta guía con tu equipo o amigos desarrolladores.
- Guárdala para tenerla siempre a mano.
- ¡Dale un like o comenta si tienes preguntas!

