

Importancia del Test Driven Development

Francisco Alfaro

29 de Septiembre del 2022



Introducción

Por qué debería usarlo

Evidencia empírica

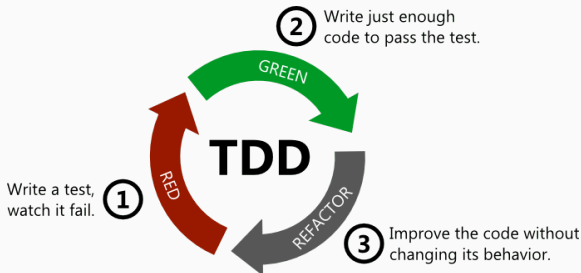
Librerías disponibles

Referencias



Qué es TDD

En su forma más simple consiste en un proceso iterativo de 3 fases:



Ejemplo Sencillo

Testear la función **paridad**, que determina si un número natural es par o no.

Lo primero que se debe hacer es crear el test, para ello se ocupará la librería `pytest`.

```
@pytest.mark.parametrize(
    "number, expected",
    [
        (2, 'par'),
    ])
def test_paridad(number, expected):
    assert paridad(number) == expected
```



El test nos dice que si el input es el número 2, la función **paridad** devuelve el output 'par'. Como aún no hemos escrito la función, el test fallará (**fase red**).

```
===== test session starts =====
platform linux -- Python 3.8.10, pytest-6.2.4, py-1.10.0, pluggy-0.13.1
rootdir: /home/fralfaro/PycharmProjects/ds_blog
plugins: anyio-3.3.0
collected 1 item

temp/test_funcion.py F                                     [100%]
===== 1 failed in 0.14s =====
```



Ahora, se escribe la función paridad (**fase green**):

```
def paridad(n:int)->str:
    """
    Determina si un numero natural es par o no.

    :param n: numero entero
    :return: 'par' si es el numero es par; 'impar' en otro caso
    """
    return 'par' if n%2==0 else 'impar'
```



Ejemplo Sencillo

Volvemos a correr el test:

```
===== test session starts =====  
platform linux -- Python 3.8.10, pytest-6.2.4, py-1.10.0, pluggy-0.13.1  
rootdir: /home/fralfaro/PycharmProjects/ds_blog  
plugins: anyio-3.3.0  
collected 1 item  
  
temp/test_funcion.py . [100%]  
===== 1 passed in 0.06s =====
```



Hemos cometido un descuido a proposito, falta testear el caso si el número fuese impar, por lo cual reescribimos el test (**fase refactor**)

```
@pytest.mark.parametrize(
    "number, expected",
    [
        (2, 'par'),
        (3, 'impar'),
    ])
def test_paridad(number, expected):
    assert paridad(number) == expected
```



Ejemplo Sencillo

y corremos nuevamente los test:

```
===== test session starts =====  
platform linux -- Python 3.8.10, pytest-6.2.4, py-1.10.0, pluggy-0.13.1  
rootdir: /home/fralfaro/PycharmProjects/ds_blog  
plugins: anyio-3.3.0  
collected 2 items  
  
temp/test_funcion.py .. [100%]  
===== 2 passed in 0.06s =====
```



Tabla de Contenidos

Introducción

Por qué debería usarlo

Evidencia empírica

Librerías disponibles

Referencias



Por qué debería usarlo

- Testear correctamente nos ayuda a clarificar los límites del problema y cómo podemos resolverlo. Con el tiempo esto ayuda a obtener un diseño modular y reusable del código.
- Escribir tests ayuda la forma en que escribimos código, haciéndolo más legible a otros desarrolladores.
- Verifica que el código funciona de la manera que se espera, y lo hace de forma automática.



Por qué debería usarlo

- Te permite realizar **refactoring** con la certeza de que no has roto nada.
- Los tests escritos sirven como documentación para otros desarrolladores.
- Es una práctica requerida en metodologías de desarrollo de software agile.



Introducción

Por qué debería usarlo

Evidencia empírica

Librerías disponibles

Referencias



El 2008, Nagappan, Maximilien, Bhat y Williams publicaron el paper llamado **Realizing Quality Improvement Through Test Driven Development - Results and Experiences of Four Industrial Teams**, en donde estudiaron 4 equipos de trabajo (3 de Microsoft y 1 de IBM), con proyectos que variaban entre las 6000 líneas de código hasta las 155k.



Algunas conclusiones:



Todos los equipos demostraron una baja considerable en la densidad de defectos: 40 % para el equipo de IBM, y entre 60-90 % para los equipos de Microsoft.



Como todo en la vida, nada es gratis:



Incremento del tiempo del desarrollo de software varía entre un 15 % a 35 %.



Sin embargo:



Desde un punto de vista de eficacia este incremento en tiempo de desarrollo se compensa por los costos de mantención reducidos debido al incremento en calidad.



¿Puedo siempre usar TDD?

La respuesta es **NO**, pero puedes usarlo casi siempre.

El análisis exploratorio es un caso en que el uso de TDD no hace sentido. Una vez que tenemos definido el problema a solucionar y un mejor entendimiento del problema podemos aterrizar nuestras ideas a la implementación vía testing.



Tabla de Contenidos

Introducción

Por qué debería usarlo

Evidencia empírica

Librerías disponibles

Referencias



- **unittest**: Módulo dentro de la librería estándar de Python. Permite realizar tests unitarios, de integración y end to end.
- **pytest**: Librería de testing ampliamente usada en proyectos nuevos de Python.
- **hypothesis**: Librería para escribir tests vía reglas que ayuda a encontrar casos borde.
- **coverage**: Herramienta para medir la cobertura de código de los proyectos.



Tabla de Contenidos

Introducción

Por qué debería usarlo

Evidencia empírica

Librerías disponibles

Referencias



- **Google Testing Blog**: En particular destaca la serie `Testing on the Toilet`.
- **Design Patterns** : Los patrones de diseño de software tienen en consideración que el código sea testeable.
- Cualquier artículo de **Martin Fowler** sobre testing.



Importancia del Test Driven Development

Francisco Alfaro

29 de Septiembre del 2022

