Testing

TB022 - Esteban - Riesgo - Kristal

Qué son las pruebas de software?

Son una actividad de control de la calidad del software construido.

Pero por sí mismas no garantizan la calidad del software. Hay prácticas de aseguramiento de la calidad (*QA = Quality Assurance*) para mejorar el proceso de desarrollo con vistas a que el producto final tenga mejor calidad: *DevOps*, *CI/CD* (integración continua/entrega continua), etc.

Tipos de Pruebas

Pruebas de verificación (pruebas técnicas)

Verificar: controlar que hayamos construido el producto tal como pretendimos construirlo (controlar que el producto funciona).

Pruebas de validación (pruebas de usuarios)

Validar: controlar que hayamos construido el producto que nuestro cliente quería (controlar que construimos el producto correcto).

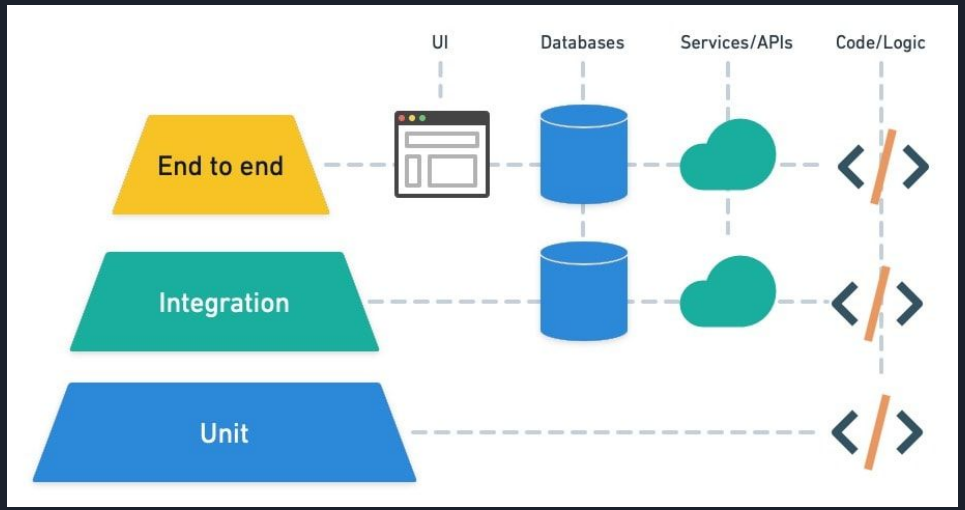
Pruebas de Verificación

Las pruebas de verificación son pruebas que los propios desarrolladores ejecutan para ver que están logrando que el programa funcione como ellos pretenden.

- **Las pruebas unitarias (*Unit testing*)** verifican pequeñas porciones de código. Por ejemplo, verifican alguna responsabilidad única de un método.
- **Las pruebas de integración (*Integration testing*)** prueban que varias porciones de código, trabajando en conjunto, hacen lo que pretendíamos. Por ejemplo, se trata de pruebas que involucran varios métodos, clases o incluso subsistemas enteros.
- **Las pruebas de regresión (*Regression testing*)** garantizan que la aplicación siga funcionando correctamente después de producirse algún cambio en el código.
- **Las pruebas de punta a punta (*E2E o End-to-End testing*)** verifican funcionalidades que atraviesan todas las capas de la aplicación, p. ej. desde la pantalla a la base de datos.
- **Las pruebas de sistema (*System testing*)** evalúan cómo los diferentes componentes interactúan juntos en la aplicación completa e integrada.

Pruebas de Verificación

Más costoso



Más barato

Más lento

Más rápido

Tendremos muchas pruebas unitarias, varias pruebas de integración y algunas pruebas e2e

Pruebas de Verificación

Hay ocasiones en que debemos probar código que necesita de otros objetos, métodos o funciones para poder funcionar: en esas situaciones se utilizan objetos ficticios o dobles de prueba (stubs, mocks, fakes, etc.), que ayudan a aislar el código a probar.

Las pruebas de verificación podrían ser de caja negra o de caja blanca:

- Decimos que una prueba es de **caja negra** cuando la ejecutamos sin mirar el código que estamos probando.
- Cuando, en cambio, analizamos el código durante la prueba, decimos que es una prueba de **caja blanca**.

En general, se prefiere **probar el funcionamiento y no verificar la calidad del código**.

assert

`assert <sentencia>`

Obtiene el valor de verdad de `<sentencia>`. Si es `True`, no hace nada. Si es `False`, levanta un AssertionError. No indica cuál es la diferencia, sólo que existe.

```
def func(a, b, c):
    ...

assert func(1,2,3) == x
assert func(1,2,3) in resultado_esperado
assert func(3,2,1) not in resultado_esperado
```

pytest

pytest es el módulo para hacer tests más común en Python

Utiliza assert para hacer validaciones pero muestra con detalle los errores

```python
def func(a, b, c):
    ...
```

```python
def test_func():
    a, b, c = 1, 2, 3
    resultado = 6
    assert func(a, b, c) == resultado
```

Cada función de test va a tener un solo *assert* (idealmente). Por cada verificación que queramos hacer, hay que hacer un test distinto.

Instalación

Debemos usar pip

```
pip install pytest
```

*Recomendamos utilizar *ambientes virtuales* para facilitar el manejo de las versiones del lenguaje y de sus paquetes.

pytest: Ejemplo

```python
def factorial(x):
    if x < 0:
        raise ValueError
    res = 1
    for i in range(2, x):
        res *= i
    return res
```

```python
def test_factorial_cero():
    assert factorial(0) == 1

def test_factorial_positivo():
    assert factorial(10) == 362880

def test_factorial_negativo():
    assert factorial(-1) == ??
```

Por qué hacer tests?

Si ya sabemos debuggear nuestro código, para qué necesito tests? Si ya debuggeamos y verificamos que anda y devuelve bien, qué gano con esto?

El código no es estático, va cambiando con el tiempo.

El código que escribimos para resolver un problema puntual luego puede ser modificado (por nueva funcionalidad, nuevos casos borde, cambio de nombre de variables o funciones siendo usadas, etc.).

Cada vez que modificamos código, debemos verificar que siga funcionando.

Tener tests nos permite escribir la verificación a realizar una única vez, y luego ejecutarla cuando necesitemos. Si agregamos nueva funcionalidad se agregarán nuevos tests, pero los existentes deberían seguir funcionando. Sino, deberíamos verificar cada vez que cada pequeña cosa ande bien.

Mientras más unitarios nuestros tests, al fallar algo, más fácil será entender qué salió mal y donde arreglarlo sin necesidad de hacer debugging extra.

Cómo elegir qué probar?

Más allá de cómo una función esté implementada, debemos definir pre y post condiciones cuando escribimos nuestras funciones.

Debemos determinar los posibles escenarios para los cuales cambiando el input, el output de la función es distinto.

En los tests de factorial, por qué no probamos con 2, 3, 4, .. si también cambia el output?

Podemos agrupar los casos a probar en:

- **Casos de error/de excepción**: todos los distintos errores que devuelva una función deben ser probados por separado.
- **Casos borde**: para los cuales el algoritmo usado usa alguna condición especial o está en el límite entre otro tipo de resultado.
- **Casos generales**: cuando el código sigue el *happy path*. Puede haber más de un camino de éxito. Sólo es necesario una prueba por cada uno de ellos.

Ejercicio: Tateti

Sobre el mismo Tateti que nos habían pasado, para evitar tener más problemas del mismo estilo en el futuro, nos pidieron que hagamos pruebas para cada una de sus funciones.

Mocks

Mocking es una estrategia de testing que nos permite “simular” comportamiento. Es útil para probar partes de código o funcionalidades específicas sin tener que levantar todo un ambiente entero o “preparar” el caso de uso.

Un "mock" es una imitación de un objeto concreto al cual podemos moldear cómo nosotros queramos para cumplir con las expectativas de nuestras pruebas. Podemos decirle cómo tiene que comportarse frente a ciertas situaciones (qué devolver cuando una función o método es invocado, qué efectos secundarios, como levantar excepciones, tiene, ver qué sentencias fueron efectivamente invocadas)

Casos de usos habituales:

- En testeos unitarios que involucran funcionalidades de diferentes puntos de código que interactúan entre sí (por ejemplo, una clase que utiliza a otra)
- Para simular interacciones con servicios o componentes externas al que estamos testeando (por ejemplo, obtener información de una página web o API)

Como regla general, solo mockeamos testeos UNITARIOS. Nunca de integración o E2E

Mocks en Python

Vamos a usar la biblioteca pytest-mock. Es bastante simple de usar, provee features relativamente básicas:

1. Todas las funciones de test que van a usar mocks recibirán un valor por parámetro de tipo `MockerFixture` el cual nos permite crear mocks y otras cosas.
2. Toda función/método a testear que tenga un dependencia, es decir, dependa de otra función u objeto para lograr su cometido, tendrá esa dependencia mockeada con un objeto de tipo `Mock`.
3. Estos mocks a crear deben tener siempre estipulado un `return_value` para el método o atributo invocados sobre ellos.
4. Al final de cada test, se debe asegurar que se hicieron las llamadas correspondientes a cada mock (es decir, cada dependencia) de la función o método siendo testeado.

Mocks: ejemplo

Simulemos una request a una página web: https://catfact.ninja/fact

Nuestra función será de la siguiente manera:

```
import requests

class CatServerError(Exception):
    pass

def get_cat_fact():
    url = "https://catfact.ninja/fact"
    response = requests.get(url)

    if response.status_code == 200:
        fact = response.json()
        return fact['fact']
    else:
        raise CatServerError("Sorry, could not fetch a cat fact at the moment.")
```

Mocks: ejemplo

Un primer test a hacer podría ser el siguiente:

```python
def test_get_cat_fact_caso_feliz(mocker: MockerFixture):
    mock_response = mocker.Mock()
    mock_response.status_code = 200
    mock_response.json.return_value = {'fact': 'Cats are cute!'}

    mocker.patch('requests.get', return_value=mock_response)

    result = get_cat_fact()

    requests.get.assert_called_once()
    assert result == 'Cats are cute!'
```

Setup

Crear el mock

Programarle comportamiento

?

Execution

Verification

Verificamos que se haya hecho la llamada que queríamos

patch vs Mock

Un `Mock` como dijimos es un objeto impostor. Simula ser otro tipo de objeto donde nosotros le decimos cómo comportarse.

Sólo sirve para objetos.

`patch` permite emparchar (o "patchear") una función o variable para que en lugar de hacer lo que sea que haga, devuelva lo que nosotros le digamos.

Son básicamente la misma idea, pero trabajan sobre distintas cosas.

A la acción de hacer un patch de una función (y también se puede hacer para objetos y sus métodos) se le dice Monkey Patch y no se aplica únicamente a testing.

Mocks: ejemplo 2

Tenemos una función que procesa un archivo de texto con formato `nombre,apellido,dni,email` y devuelve un diccionario `dni->email` sólo de aquellas personas que son son extranjeros

```python
class ArchivoNoEncontrado(Exception):
    pass

def obtener_extranjeros(path: str) -> dict[str, str]:
    padron = {}

    try:
        with open(path, "r") as archivo:
            for linea in archivo:
                datos = linea.rstrip("\n").split(",")

                nombre, apellido, email, dni = datos

                # Filtrar DNIs que empiecen con 9 (extranjeros)
                if dni.startswith("9"):
                    padron[dni] = email

    except FileNotFoundError:
        raise ArchivoNoEncontrado("no se encontró el archivo")

    return padron
```

Qué casos debemos probar?

- Que si el archivo no existe se levante la excepción correcta con el mensaje correcto
- Que si el archivo existe se procesen los datos correctamente (chequear que los datos devueltos sean correctos)

Necesitamos mockear el llamado a leer el archivo

Mocks: ejemplo 2

Tenemos una función que procesa un archivo de texto con formato `nombre,apellido,dni,email` y devuelve un diccionario `dni->email` sólo de aquellas personas que son son extranjeros

```python
def test_file_not_found(mocker):
    mocker.patch("builtins.open", side_effect=FileNotFoundError)
    with pytest.raises(ArchivoNoEncontrado) as e:
        obtener_nacionalizados("archivo_inexistente.txt")

        assert str(e.value) == "no se encontró el archivo"
```

```python
def test_file_with_correct_values(mocker):
    contenido = """Juan,Perez,juan@gmail.com,91234567
        Ana,Gomez,ana@mail.com,21234567
        Luis,Lopez,luis@gmail.com,93456789
        """
    mock_open = mocker.patch("builtins.open",
mocker.mock_open(read_data=contenido))
    resultado = obtener_nacionalizados("ejemplo.txt")

    assert len(resultado) == 2
    assert resultado["91234567"] == "juan@gmail.com"
    assert resultado["93456789"] == "luis@gmail.com"
    assert "21234567" not in resultado

    mock_open.assert_called_once_with("ejemplo.txt", "r")
    mock_open.assert_called_once_with("ejemplo.txt", "r")
```

Coverage

Una vez que se corrieron todas las pruebas, se puede hacer un análisis de coverage para ver cuántas y cuáles líneas de código del segmento que estamos testeando fueron ejecutadas por nuestras pruebas.

```
$ coverage run -m pytest
```

Generará un archivo HTML con el detalle

Un porcentaje alto de coverage **NO** implica que nuestro código esté bien testeado. Ninguna herramienta nos asegura eso.

Es por eso que debemos analizar en detalle todos los posibles casos a probar nosotros mismos. El *coverage* nos puede ayudar luego de eso para ver si nos faltó considerar algún caso viendo qué código no fue ejecutado.

El coverage es un tipo de *análisis estático de código*

Análisis estático de código

Hay muchas herramientas distintas de este tipo.

Se usan tanto para validar la calidad del código o como para mejorarlo.

Ademas de linters/formatters y coverage checkers hay también:

- Type checkers: verifican que el tipo de las estructuras sea válido. Se usan en lenguajes no tipados (como Python)
- Complexity analyzers: ven que tan “complejo” es el código
- Dependency checkers: chequeos continuos de dependencias para ver que no haya ninguna versión desactualizada
- Duplicate code detectors
- Security Scanners: detecta si alguna biblioteca utilizada ha sufrido alguna vulnerabilidad

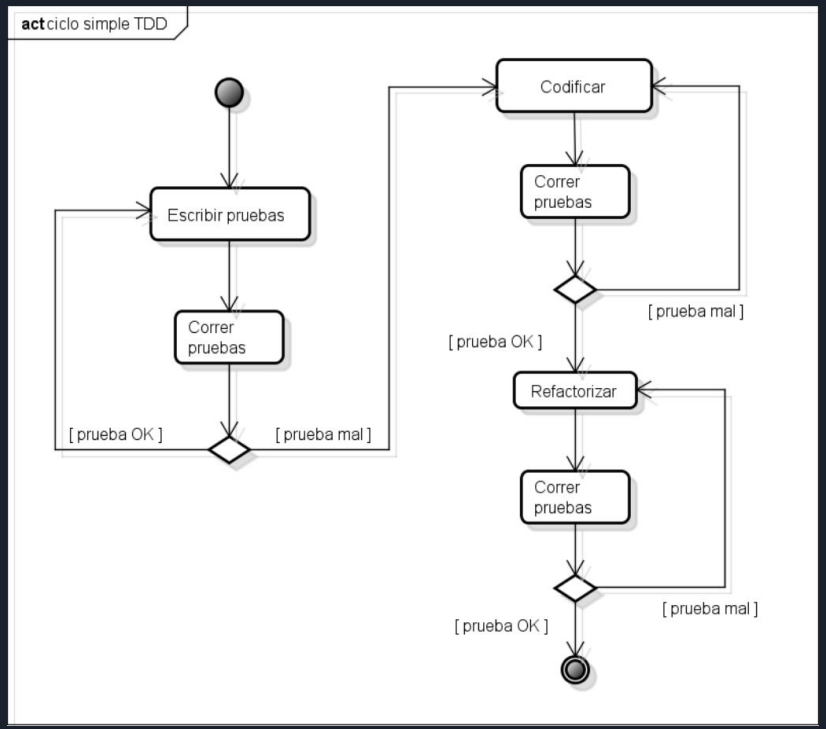
TDD: Test Driven Development

Fue la primera práctica basada en automatización de pruebas bien soportada por herramientas. Fue presentada por Kent Beck en el marco de Extreme Programming y enseguida surgieron frameworks para llevarla a la práctica.

TDD incluye tres sub prácticas:

- **Automatización**: las pruebas del programa deben ser hechas en código (generalmente siguiendo las reglas "arrange, act, assert" (también llamadas "setup, execution, verification") o "given, when, then"), y con la sola ejecución del código de pruebas debemos saber si lo que estamos probando funciona bien o mal.
- **Test-First**: las pruebas se escriben antes del propio código a probar.
- **Refactorización posterior**: para mantener la calidad del código, se lo cambia sin cambiar la funcionalidad, manteniendo las pruebas como reaseguro.

Ciclo de TDD


act ciclo simple TDD
Escribir pruebas
Correr pruebas
[prueba OK]
[prueba mal]
Codificar
Correr pruebas
[prueba mal]
[prueba OK]
Refactorizar
Correr pruebas
[prueba mal]
[prueba OK]

Ventajas de TDD

- Las pruebas en código sirven como documentación del uso esperado de lo que se está probando, sin ambigüedades.
- Las pruebas escritas con anterioridad ayudan a entender mejor lo que se está por desarrollar.
- Las pruebas escritas con anterioridad suelen incluir más casos de pruebas negativas que las que escribimos a posteriori.
- Escribir las pruebas antes del código a probar minimiza el condicionamiento del autor por lo ya construido. También da más confianza al programador sobre el hecho de que el código que escribe siempre funciona.
- Escribir las pruebas antes del código a probar permite especificar el comportamiento sin restringirse a una única implementación.
- La automatización permite independizarse del factor humano y facilita la repetición de las mismas pruebas a un costo menor.
- La refactorización constante facilita el mantenimiento de un buen diseño a pesar de los cambios que, en caso de no hacerla, lo degradarían.

Fin