UNIT TESTING

Introduzione al Unit Testing

- Verifica del corretto funzionamento di singole unità di codice
- Un'unità è tipicamente una funzione, un metodo o una classe
- Isolato dalle dipendenze esterne (database, API, filesystem)
- Automatizzato e ripetibile
- Benefici:
  - Documentazione eseguibile
  - Prevenzione di regressioni
  - Miglioramento del design del codice
  - Confidenza nelle modifiche
- Framework popolari:
  - Python: unittest, pytest
  - JavaScript: Jest, Mocha

Anatomia di un Unit Test

- **Setup**: preparazione dell'ambiente di test
- **Esecuzione**: chiamata alla funzione o al metodo da testare
- **Verifica**: controllo che i risultati siano quelli attesi
- **Teardown**: pulizia dell'ambiente dopo il test

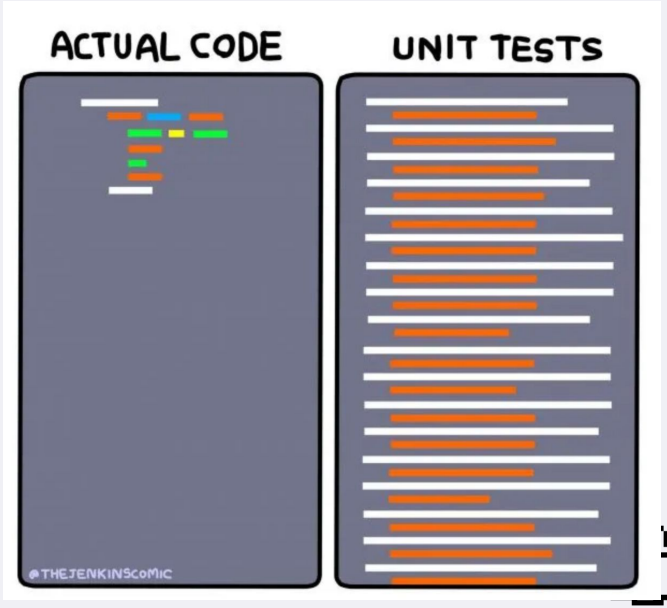
Perché fare Unit Testing?

Vantaggi immediati

- Individuazione precoce dei bug
- Facilita il refactoring e la manutenzione

Vantaggi a lungo termine

- Maggiore fiducia nel codice
- Riduzione del "debito tecnico"
- Sviluppo più rapido di nuove funzionalità
- Onboarding più semplice per nuovi sviluppatori



Principi del buon Unit Testing

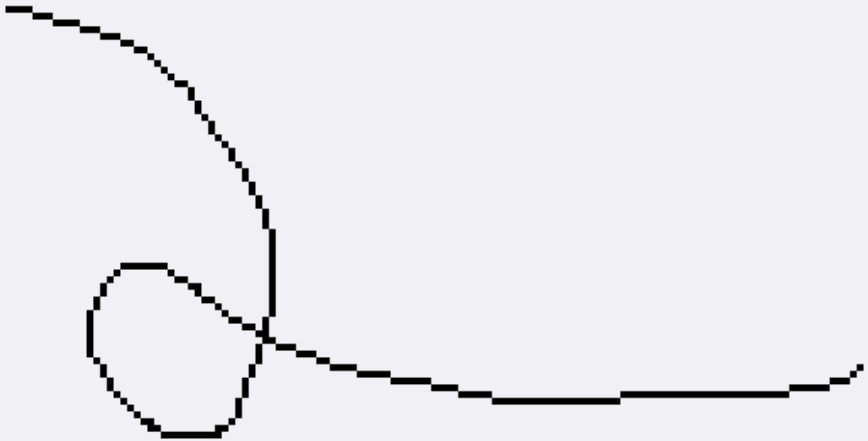
FIRST

- **F**ast: i test devono essere veloci
- **I**ndependent: ogni test deve essere indipendente dagli altri
- **R**epeatable: i test devono dare lo stesso risultato ad ogni esecuzione
- **S**elf-Validating: i test devono determinare autonomamente se hanno successo o no
- **T**imely: i test dovrebbero essere scritti prima o insieme al codice

Le 3 A

- **Arrange**: preparazione dell'ambiente
- **Act**: esecuzione del codice da testare
- **Assert**: verifica dei risultati

UNIT TESTING CON Python



Unit Test in Python con unittest

```python
# file: calcoli.py
def somma(a, b):
    return a + b

def moltiplica(a, b):
    return a * b

def dividi(a, b):
    if b == 0:
        raise ValueError("Divisione per zero non consentita")
    return a / b
```

```python
# file: test_calcoli.py
import unittest
from calcoli import somma, moltiplica, dividi

class TestCalcoli(unittest.TestCase):

    def test_somma(self):
        self.assertEqual(somma(5, 3), 8)
        self.assertEqual(somma(-1, 1), 0)
        self.assertEqual(somma(0, 0), 0)

    def test_moltiplica(self):
        self.assertEqual(moltiplica(4, 3), 12)
        self.assertEqual(moltiplica(0, 5), 0)
        self.assertEqual(moltiplica(-2, 3), -6)

    def test_dividi(self):
        self.assertEqual(dividi(10, 2), 5)
        self.assertEqual(dividi(0, 5), 0)
        self.assertAlmostEqual(dividi(5, 3), 1.6666666666666667)

    def test_dividi_zero(self):
        with self.assertRaises(ValueError):
            dividi(10, 0)

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

unittest documentazione

Method	Checks that
`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`
`assertTrue(x)`	`bool(x) is True`
`assertFalse(x)`	`bool(x) is False`
`assertIs(a, b)`	`a is b`
`assertIsNot(a, b)`	`a is not b`
`assertIsNone(x)`	`x is None`
`assertIsNotNone(x)`	`x is not None`
`assertIn(a, b)`	`a in b`
`assertNotIn(a, b)`	`a not in b`
`assertIsInstance(a, b)`	`isinstance(a, b)`
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`

Unit Test in Python con pytest

```python
# file: calcoli.py
def somma(a, b):
    return a + b

def moltiplica(a, b):
    return a * b

def dividi(a, b):
    if b == 0:
        raise ValueError("Divisione per zero non consentita")
    return a / b
```

```python
# Non è necessario ereditare da classi specifiche!
# Installazione: pip install pytest

# La stessa classe Calculator definita prima
class Calculator:
    def add(self, a, b):
        return a + b

    def divide(self, a, b):
        if b == 0:
            raise ValueError("Impossibile dividere per zero")
        return a / b

# I test con pytest sono più semplici e leggibili
def test_add():
    calc = Calculator()
    assert calc.add(2, 3) == 5

def test_divide():
    calc = Calculator()
    assert calc.divide(10, 2) == 5

import pytest

def test_divide_by_zero():
    calc = Calculator()
    with pytest.raises(ValueError):
        calc.divide(10, 0)
```

UNIT TESTING CON Javascript

Unit Testing in JavaScript/Node.js (Jest)

```
// Installazione: npm install --save-dev jest

// La classe che vogliamo testare - calculator.js
class Calculator {
  add(a, b) {
    return a + b;
  }

  divide(a, b) {
    if (b === 0) {
      throw new Error("Impossibile dividere per zero");
    }
    return a / b;
  }
}

module.exports = Calculator;
```

```
// Il file di test - calculator.test.js
const Calculator = require('./calculator');

describe('Calculator', () => {
  let calc;

  beforeEach(() => {
    // Questo viene eseguito prima di ogni test
    calc = new Calculator();
  });

  test('should add two numbers correctly', () => {
    // Arrange e Act
    const result = calc.add(2, 3);

    // Assert
    expect(result).toBe(5);
  });

  test('should divide two numbers correctly', () => {
    const result = calc.divide(10, 2);
    expect(result).toBe(5);
  });

  test('should throw error when dividing by zero', () => {
    // Verifichiamo che una funzione sollevi un'eccezione
    expect(() => calc.divide(10, 0)).toThrow("Impossibile dividere per zero");
  });
});
```

- **Avvio del testing con:**
  a. `npm run test`

```
npm i jest-cli -g
jest
```

Mocha (test runner) e Chai (libreria di asserzioni)

```
// Installazione: npm install --save-dev jest

// La classe che vogliamo testare - calculator.js
class Calculator {
  add(a, b) {
    return a + b;
  }

  divide(a, b) {
    if (b === 0) {
      throw new Error("Impossibile dividere per zero");
    }
    return a / b;
  }
}

module.exports = Calculator;
```

- **Avvio del testing con:**
  a. `npm run test`

```
// Installazione: npm install --save-dev mocha chai

// Il file di test con Mocha e Chai
const Calculator = require('./calculator');
const expect = require('chai').expect;

describe('Calculator', function() {
  let calc;

  beforeEach(function() {
    calc = new Calculator();
  });

  it('should add two numbers correctly', function() {
    const result = calc.add(2, 3);
    expect(result).to.equal(5);
  });

  it('should divide two numbers correctly', function() {
    const result = calc.divide(10, 2);
    expect(result).to.equal(5);
  });

  it('should throw error when dividing by zero', function() {
    // Verifichiamo che una funzione sollevi un'eccezione
    expect(function() {
      calc.divide(10, 0);
    }).to.throw("Impossibile dividere per zero");
  });
});
```

Best Practices e altro

Best Practices

Cosa testare

- Logica di business complessa
- Edge cases e gestione degli errori
- Funzionalità critiche dell'applicazione

Cosa non testare

- Codice di terze parti
- Getter e setter semplici
- Implementazioni banali

Organizzazione dei test

- Struttura dei file di test (mirror della struttura del codice)
- Naming convenzionale (test_*.py, *.test.js, *_test.go)
- Raggruppamento logico dei test

Test-Driven Development (TDD)

Il ciclo Red-Green-Refactor

1. **Red:** Scrivi un test che fallisce
2. **Green:** Scrivi il minimo codice necessario per far passare il test
3. **Refactor:** Migliora il codice mantenendo i test in verde

Esempio di TDD in Python

```python
# 1. RED - Scriviamo un test che fallisce
def test_validate_password():
    assert validate_password("abc123") == False  # troppo corto
    assert validate_password("abcdefgh") == False  # nessun numero
    assert validate_password("12345678") == False  # nessuna lettera
    assert validate_password("abcd1234") == True  # valida

# Il test fallirà perché la funzione non esiste ancora

# 2. GREEN - Implementiamo la funzione per far passare il test
def validate_password(password):
    if len(password) < 8:
        return False
    if not any(c.isdigit() for c in password):
        return False
    if not any(c.isalpha() for c in password):
        return False
    return True

# 3. REFACTOR - Miglioriamo il codice mantenendo i test in verde
def validate_password(password):
    has_min_length = len(password) >= 8
    has_digit = any(c.isdigit() for c in password)
    has_letter = any(c.isalpha() for c in password)

    return has_min_length and has_digit and has_letter
```

Conclusioni e Risorse

Riassunto

- I test unitari sono fondamentali per la manutenzione del codice
- Python e JavaScript offrono diversi framework con approcci simili
- Il TDD è una pratica che migliora la qualità del codice

Passaggi successivi

- Integrare i test in CI/CD
- Esplorare altre tipologie di test (integrazione, sistema, end-to-end)
- Misurare la copertura del codice

Risorse utili

- Documentazione ufficiale unittest
- Documentazione Jest


FINE
WWW.DUCC10.ME