TDD prime impressioni

Davide Deponti

WE TEST BEHAVIOUR NOT IMPLEMENTATION

Un po di storia...

Tanto tempo fa, c'erano i programmatori che scrivevano codice e quando avevano terminato se avanzava tempo scrivevano anche dei test automatici per verificare il codice (bhè succede anche ora)

Col passare degli anni ci si è resi sempre più conto di quanto fossero importanti i test, erano veramente utili a qualcosa!

Fu così che un bel giorno qualcuno disse: "Ma se scriviamo prima i test e dopo il codice? E se facciamo in modo che il codice si orientato esclusivamente all'obiettivo di solo a far passare i test?"

Quel qualcuno era **Kent Beck**

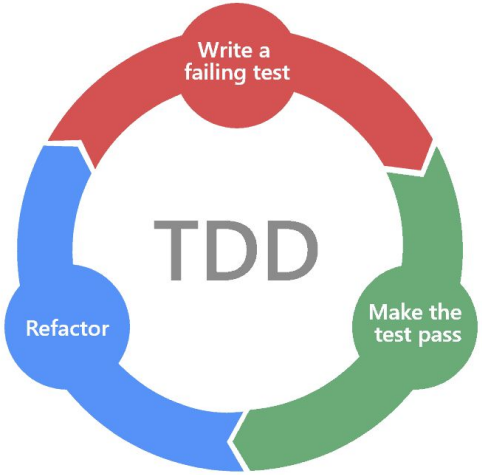
E più o meno questa è la nascita del Test Driven Development (TDD)

Cos'è il TDD

Il TDD si basa sull'idea che i test debbano essere scritti prima di scrivere il codice effettivo.

Le 3 regole principali del TDD sono:

- RED: scriviamo prima i test che fallisce
- GREED: scriviamo il codice di produzione che fa passare il test
- REFACTOR: miglioriamo il codice di produzione facendo attenzione a far passare sempre i test



"Any fool can write code that a computer can understand. Good programmers write code that humans can understand" Martin Fowler

Nel Dettaglio

Quando iniziamo ad usare il TDD potremmo trovarci spaesati, dobbiamo tenere a mente 3 step fondamentali

Fake implementation
Iniziamo con lo scrivere il codice con gli esatti valori che servono a far passare il test

Obvious implementation
Successivamente scrivi il codice ovvio, l'obbiettivo in questa fase è scrivere meno codice possibile non codice generico perchè ancora non sei in grado

Triangulation with the next test
Ora potresti voler generalizzare il tuo componente, ma ancora non sai come. Scrivi altri test e pian piano avrai modo di capire come generalizzare il tuo codice (sempre in ottica di far passare i test e di scrivere meno possibile).

Requisito: una funzione che dati 2 numeri in ingresso restituisce la somma

Caso d’uso in breve

1 Scrivo TEST

```
test('Somma di due numeri positivi', () => {
  expect(somma(2, 3)).toBe(5);
});
```

2 Eseguo test che va in Failed

3 Scrivo fake implementation

```
function somma(a, b) {
  return 5;
}
```

4 Eseguo test che va in Success

5 Scrivo l’implementazione più ovvia:

```
function somma(a, b) {
  return a + b;
}
```

6 Eseguo test che va in Success

7…ora posso scrivere altri test per altre casistiche ed eventualmente generalizzare ed evolvere il codice

E non dimentichiamoci mai di committare ad ogni nuovo test success e ogni refactoring

Potenziali punti negativi

Prestazioni
Lo svantaggio del codice ovvio può essere la riduzione delle prestazioni o comunque il fatto che non ne tiene conto

Sforzo iniziale eccessivo per piccoli progetti
TDD potrebbe non essere sempre giustificato per progetti molto piccoli o rapidi prototipi, poiché l'overhead iniziale potrebbe superare i benefici.

Costi di manutenzione dei test obsoleti
Nel corso del tempo, i test obsoleti o non più rilevanti possono accumularsi e richiedere manutenzione, aumentando il carico di lavoro generale.

Difficoltà nell'applicare a tutto
Il TDD è particolarmente efficace per la logica di business e le componenti di basso livello, ma può essere più difficile da applicare alle interfacce utente e ad alcuni aspetti dell'architettura.

Resistenza al cambiamento
Alcuni sviluppatori potrebbero resistere all'adozione di TDD, poiché richiede un cambiamento culturale e comportamentale significativo nel modo in cui scrivono il codice.

Conclusione

Dopo aver descritto il TDD e anche alcune problematiche che possono derivare dall' applicarlo concludiamo con una carrellata di benefici che si porta dietro

- Migliora la qualità del software
- Migliora la progettazione del codice
- Il codice è meglio documentato
- Refactoring più sicuri
- Codice più stabile
- Debug molto più semplice
- Si evita di fare sovra ingegnerizzazione del codice
- Si risparmia tempo nel lungo termine