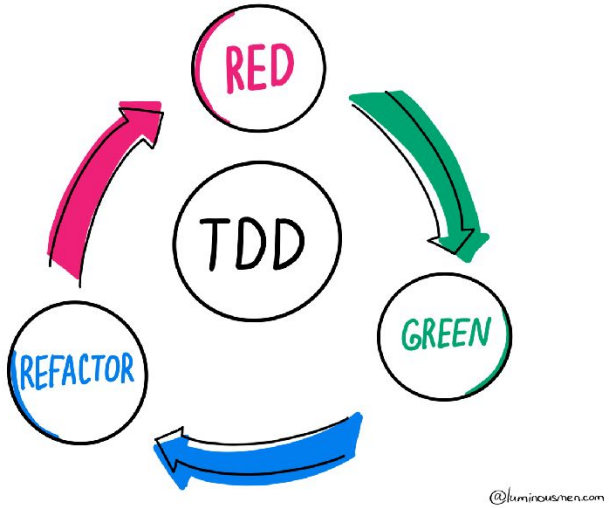
Test Driven Development

Che cos’è?

Il **Test Driven Development** (**TDD**) è un modello di sviluppo del software che prevede che la stesura dei test automatici avvenga prima di quella del software che deve essere sottoposto a test, e che lo sviluppo del software applicativo sia orientato esclusivamente all'obiettivo di passare i test automatici precedentemente predisposti.

Più in dettaglio, il TDD prevede la ripetizione di un breve ciclo di sviluppo in tre fasi, detto "ciclo TDD".

L'invenzione del metodo si deve a **Kent Beck**, uno dei padri dell'**extreme programming** (**XP**) e delle metodologie agili.



eXtreme Programming practices

La **programmazione estrema** è una metodologia di sviluppo del software mirata a migliorare la qualità del codice e la responsività al cambiamento dei requisiti del cliente.

In quanto tipo di metodologia di sviluppo agile, richiede uno sviluppo in cicli brevi con pubblicazioni frequenti, con lo scopo di migliorare la produttività e introdurre punti di controllo nei quali i nuovi requisiti possono essere adottati.

Altri elementi della programmazione estrema comprendono:

- Programmazione in coppia (**pair programming**)
- Estesa **revisione** di codice
- **Test unitari** del codice
- Non lavorare su funzionalità finché non sono **necessarie**
- **Semplicità** e **chiarezza** del codice
- Aspettarsi **cambiamenti dei requisiti** con il passare del tempo e con la migliore comprensione del problema
- Importanza data alla **comunicazione** diretta e frequente **tra sviluppatori e cliente** e fra gli sviluppatori stessi.

La fase rossa

Nel TDD, lo sviluppo di una nuova funzionalità comincia sempre con la stesura di un test automatico volto a validare quella funzionalità, ovvero verificare se il software la esibisce.

Poiché l'implementazione non esiste ancora, la stesura del test è un'attività creativa, in quanto il programmatore deve stabilire in quale forma la funzionalità verrà esibita dal software e comprenderne e definirne i dettagli.

Perché il test sia completo, deve essere eseguibile e, quando viene eseguito, produrre un **esito negativo**.

In molti contesti, questo implica che debba essere realizzato una **bozza minimale** del codice da testare, necessario per garantire la compilazione e l'esecuzione del test. Una volta che il nuovo test è completo e può essere eseguito, dovrebbe **fallire**. La fase rossa si conclude quando c'è un nuovo test che può essere eseguito e fallisce.

La fase verde

Nella fase successiva, il programmatore deve scrivere la **quantità minima di codice** necessaria per passare il test che fallisce. Non è richiesto che il codice scritto sia di buona qualità, elegante, o generale; l'unico obiettivo esplicito è che funzioni, ovvero **passi** il test. In effetti, è esplicitamente vietato dalla pratica del TDD lo sviluppo di parti di codice non strettamente finalizzate al superamento del test.

Quando il codice è pronto, il programmatore esegue nuovamente **tutti** i test disponibili sul software modificato (non solo quello che precedentemente falliva). In questo modo, il programmatore ha modo di rendersi conto immediatamente se la nuova implementazione ha causato fallimenti di test preesistenti, ovvero ha causato regressioni o peggioramenti nel codice. La fase verde termina quando tutti i test vengono **passati** con successo.

Refactoring

Quando il software passa tutti i test, il programmatore dedica una certa quantità di tempo a farne **refactoring**, ovvero a **migliorarne la struttura** attraverso un procedimento basato su piccole modifiche controllate volte a eliminare o ridurre difetti oggettivamente riconoscibili nella struttura interna del codice. Esempi tipici di azioni di refactoring includono:

- La scelta di identificatori più espressivi
- Eliminazione di codice duplicato
- Semplificazione dell'architettura del sorgente

L'obiettivo del refactoring non è quello di ottenere del codice "perfetto", ma solo di migliorarne la struttura, secondo la cosiddetta "regola dei Boy Scout": "lascia l'area dove ti sei accampato più pulita di come l'hai trovata".

Dopo ciascuna azione di refactoring, i test automatici vengono nuovamente eseguiti per accertarsi che le modifiche eseguite non abbiano introdotto errori.

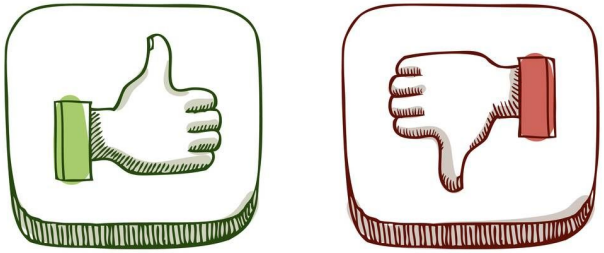
Stile di sviluppo

Il principio fondamentale del TDD è che lo sviluppo vero e proprio deve avvenire solo allo scopo di passare un test automatico che fallisce. In particolare, questo vincolo è inteso a **impedire** che il programmatore sviluppi **funzionalità non esplicitamente richieste**, e che il programmatore introduca complessità eccessiva in un progetto, per esempio perché prevede la necessità di generalizzare l'implementazione in un futuro più o meno prossimo.

I cicli TDD sono intesi come cicli di breve durata, al termine di ciascuno dei quali il programmatore ha realizzato un piccolo incremento di prodotto (con i relativi test automatici).

L'applicazione reiterata del refactoring al termine di ogni ciclo ha lo scopo di creare codice di alta qualità e buone architetture in modo incrementale, tenendo però separati l'obiettivo di costruire software funzionante (fase verde) e quello di scrivere "buon codice" (fase grigia). La breve durata dei cicli TDD tende anche a favorire lo sviluppo di componenti di piccole dimensioni e ridotta complessità.

Vantaggi



Ecco una breve lista dei benefici più importanti:

1. **Focus sui punti più importanti**: ti verrà richiesto di scomporre il problema in piccole parti, questo ti aiuterà a mantenere l'attenzione sulle cose più importanti. Il pre-requisito è proprio scomporre un grosso task in piccoli passi e sviluppare tramite unit test.
2. **Gestire compiti più semplici**: lavorando su singoli task si semplifica la risoluzione dei problemi e si accelera il processo di sviluppo. Non ti ritroverai, dopo aver scritto tutto il codice, a scoprire che qualcosa non va e non sapere perché.
3. **Integrazione semplificata**: quando le singole parti del progetto saranno completate, metterle insieme sarà un vero piacere. In caso di regressione saprai in anticipo in quale parte del codice si nasconde l'errore.
4. **Test gratis**: una volta che il task è terminato, rimangono a capitale tanti unit test che possono essere usati come base di partenze per altri unit test ed integration test, utili per migliorare la qualità del codice ed evitare regressioni.