Best Practices adottate

Gruppo: Alessio Arcara, Alessia Crimaldi, Davide Fermi, Matteo Sacco

Progetto SWENG - a.a. 2022/2023

1 Introduzione

Per l'esecuzione del progetto *Card Exchange*, è stata decisione comune e volontà del team l'aggiunta delle seguenti pratiche di Extreme Programming al framework di Scrum:

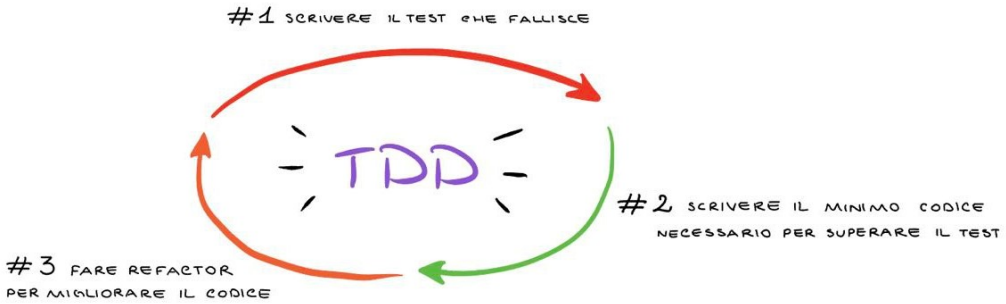
- Test Driven Development (TDD)
- Collective Code Ownership
- Pair Programming
- Continuous Integration (CI)

Inoltre, è stata aggiunta e messa in atto una pratica personalizzata ed ideata da zero dal development team denominata "Royal Rumble Review".

2 Spiegazione delle pratiche adottate

2.1 Test Driven Development (TDD)

Il Test Driven Development (TDD) è una pratica di sviluppo del software, ma fondamentalmente è uno stile di programmazione, che prevede la scrittura di test automatici prima della scrittura del codice del prodotto finale. In altre parole, il TDD segue un ciclo di sviluppo che inizia con la scrittura di un test di unità automatico, che inizialmente deve fallire in quanto il codice del prodotto finale non è ancora stato scritto. Successivamente, si aggiunge il minimo codice necessario per superare il test, ed infine il test viene eseguito per verificare che il codice del prodotto funzioni come previsto. Come ultima fase, viene valutato se il codice scritto sia migliorabile dal punto di vista strutturale, e quindi se è possibile effettuare un refactor del codice in maniera tale da migliorarne la qualità. Questo prestando però sempre attenzione a non rompere i test già presenti, poiché si romperebbero i requisti funzionali, che vanno rispettati. Tale processo si ripete quindi per ogni riga di codice che si scrive, avendo cura di collezionare tutti i test.



Il TDD, pertanto, consente di ottenere un codice di alta qualità grazie alla continua verifica delle funzionalità che si stanno sviluppando, riducendo il rischio di introdurre bug e semplificando il processo di debug. Inoltre, poiché ogni funzionalità è testata in modo isolato, il codice risulta più modulare e meno dipendente da altri componenti del sistema. Inoltre, naturalmente il TDD porta ad avere un'alta copertura del codice prodotto.

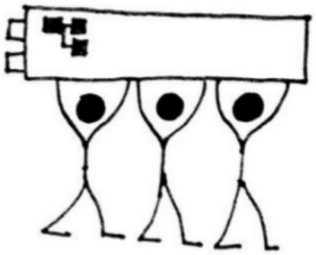
L'applicazione di questa pratica ha richiesto una conoscenza approfondita degli strumenti di test JUnit ed EasyMock utilizzati, ed una rigorosa adesione alla prassi di scrivere i test prima del codice, in quanto la progettazione è ex post.

2.2 Collective Code Ownership

La Collective Code Ownership (in italiano, "Proprietà Collettiva del Codice") è una pratica di sviluppo del software in cui l'intero team di sviluppo condivide la responsabilità del codice.
In un ambiente di sviluppo con la Collective Code Ownership, ogni membro del team ha il diritto e la responsabilità di modificare il codice esistente, correggere eventuali problemi e aggiungere nuove funzionalità. Si evita quindi così:

- la creazione di una dipendenza verso un solo individuo, che potrebbe essere fonte di impedimenti futuri;
- la frammentazione di conoscenza tra i diversi membri del team.

Pertanto, gli sviluppatori devono essere in grado di collaborare efficacemente, comunicare chiaramente e rispettare le opinioni degli altri membri del team per garantire che il codice sia sviluppato in modo coerente.



In sintesi, la Collective Code Ownership promuove la collaborazione e la responsabilità condivisa tra tutti i membri del team garantendo una maggiore efficienza nello sviluppo del software.

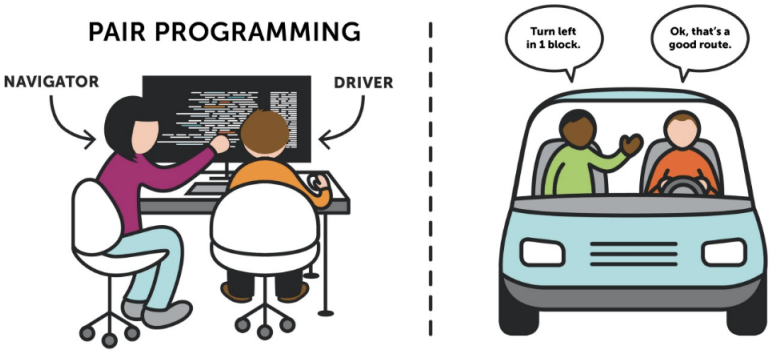
2.3 Pair Programming

Il pair programming è una pratica di sviluppo software in cui due programmatori lavorano insieme sullo stesso computer, condividendo una tastiera ed uno schermo. Questa pratica è stata sviluppata per migliorare la qualità del codice e aumentare la produttività, ma soprattutto per condividere le conoscenze tra gli sviluppatori in modo tale da raggiungere una conoscenza complessiva condivisa da tutti, raggiungendo così anche l'autonomia nello scrivere il codice. Infatti, il pair programming è particolarmente utile per i nuovi membri del development team, che possono apprendere dai colleghi più esperti attraverso questa metodologia collaborativa. Inoltre, il pair programming ha anche il vantaggio di ridurre il rischio di errori e la necessità di review future del codice, il che porta a un maggiore risparmio di tempo e ad una migliore qualità del codice.
Infine, il pair programming ha grande rilevanza in quanto può anche migliorare la soddisfazione del lavoro e la motivazione dei membri del team, poiché la collaborazione può essere una fonte di stimolo e crescita personale.
Le metodologie per fare pair programming sono diverse e varie. Qui ci limitiamo a citare le principali:

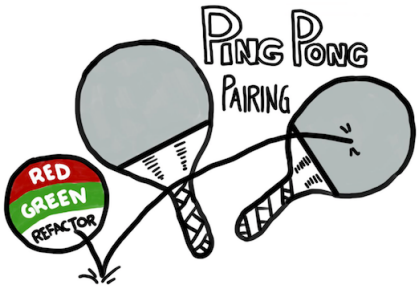
- **Driver and Navigator**. Si tratta della metodologia di pair programming più comune. Lo sviluppatore che detiene il ruolo di driver, è colui che guida, che tiene in mano la tastiera e che scrive codice pensando ad eseguire un mini goal deciso inizialmente. È importante che questo lo faccia parlando e descrivendo tutto quello che fa. Il navigator, invece, osserva e controlla il codice che sta venendo scritto dando direzioni e scambiando idee, ma senza interrompere il flow del driver, che dev'essere libero di poter fare i propri pensieri. Inoltre, il navigator deve pensare a medio-lungo termine, ossia pensare ai prossimi goal da fare e

controllare se ci sono delle problematiche a medio-lungo termine del codice che sta venendo scritto. Una volta che il primo mini goal è stato completato, si può quindi discutere di ciò che è stato prodotto.

Si procede quindi per mini goal fino ad arrivare alla conclusione di un task inizialmente suddiviso. Durante la sessione di lavoro, in questa modalità, i ruoli devono essere scambiati spesso.



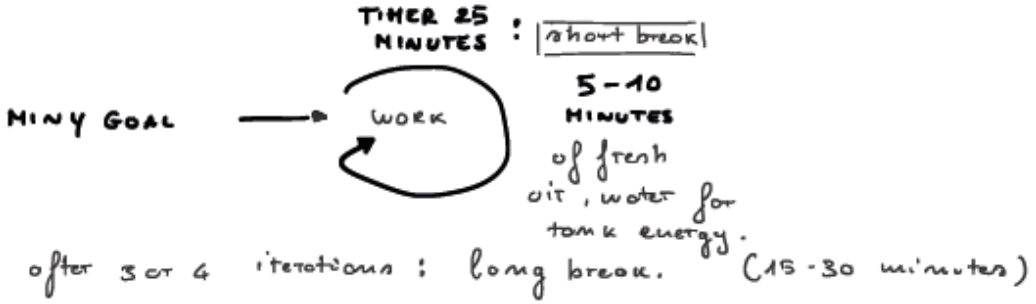
- **Ping-pong**. Si tratta di una metodologia di pair programming basata sull'approccio Test-Driven-Design. Come suggerisce il nome, essa consiste in uno scambio continuo tra i due sviluppatori: lo sviluppatore A scrive un test (il "ping"), lo sviluppatore B scrive il codice per superare il test (il "pong"), quindi scrive un nuovo test per lo sviluppatore A, e così via. In questo modo ci si scambiano i ruoli repentinamente tra il "ping" ed il "pong". Inoltre, dopo il "pong", i due sviluppatori assieme possono fare refactor. Questo modo di procedere è molto dinamico, è focalizzato sulla programmazione, aiuta a garantire che il codice scritto soddisfi le specifiche del progetto e a prevenire errori nel codice.



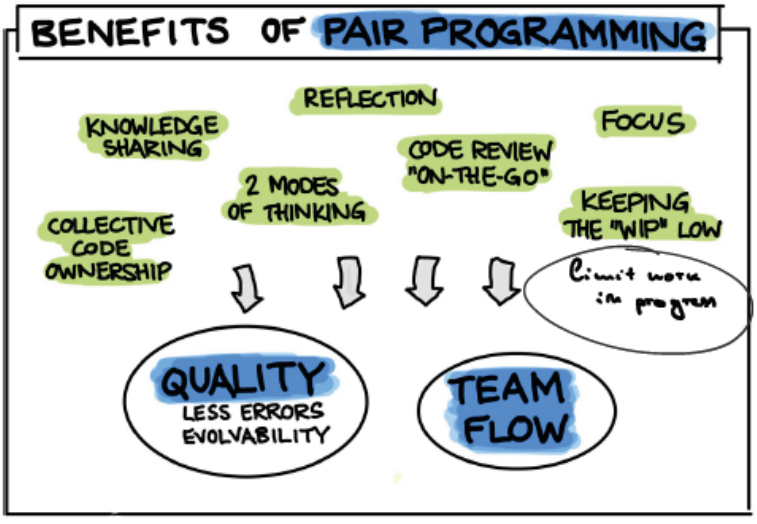
- **Strong pairing**. Si tratta di una metodologia basata semplicemente sulla metodologia di Driver and Navigator. In questo caso specifico, però, uno sviluppatore è un novizio (driver), mentre il secondo è un esperto (navigator). Questa metodologia si basa quindi sull'imparare facendo: *"learning by doing over learning by watching"*. Il navigator ha qui il ruolo di

imporre direzioni e scelte al driver, il quale esegue ed eventualmente, solo al completamento del mini goal, fa domande al driver. Come per la metodologia Driver and Navigator, si procede in questo modo fino alla fine del task.
Questo metodologia è vantaggiosa in quanto è utile alla diffusione della conoscenza del codice. Per chiari motivi, non ne va fatto però abuso, in quanto in tal caso non starebbe funzionando la condivisione di conoscenza. Per evitare ciò, è importante che ci sia una rotazione dei ruoli.

La gestione del tempo. I mini goal devono essere perseguiti per circa 25 minuti, dopo i quali dev'essere fatta una pausa di 5-10 minuti. Questo è necessario in quanto la pratica di pair programming è faticosa ed una pausa è importante per ridare nuovamente energia al pair programming. Dopo tre o quattro iterazioni di questo tipo, si deve fare una pausa più lunga di 15-30 minuti. Pertanto, in totale il pair programming non deve durare più di un paio d'ore.



Benefici del pair programming. La pratica del pair programming è potente soprattutto in quanto permette di arrivare alla **Collective Code Ownership**, ovvero alla situazione in cui tutti gli elementi del team hanno la responsabilità del codice. Questo permette ad ogni sviluppatore di avere un'autonomia nell'apportare modifiche ovunque nel codice, evitando pertanto di creare una dipendenza verso un singolo sviluppatore.



In generale, la scelta di quale modalità di pair programming applicare dipende dalle esigenze del team e dalle conoscenze tra i due sviluppatori: quelle scelte ed utilizzate dal nostro team sono state la modalità *Driver-Navigator* e la modalità *Ping-pong*.

2.4 Continuous Integration (CI)

La Continuous Integration (CI) è una pratica di sviluppo del software che prevede l'integrazione frequente del codice dei vari sviluppatori in un repository condiviso. Questo consente di rilevare gli errori in modo tempestivo e di ridurre il rischio di conflitti durante la fase di integrazione.
Per automatizzare il processo di build e di esecuzione dei test per ogni Pull Request, garantendo così il corretto funzionamento del software, abbiamo utilizzato le GitHub Actions.
Tuttavia, se il branch rimane troppo a lungo diversificato rispetto al branch principale, aumenta il rischio di conflitti e diventa necessario un lavoro aggiuntivo per sincronizzare i due rami di sviluppo.

2.5 Royal Rumble Review

Come già anticipato, si tratta di una pratica personalizzata ed ideata da zero dal team di sviluppo, che ogni sviluppatore deve applicare: da qui il nome "Royal Rumble". La *Royal Rumble Review*, infatti, prende il nome da un evento della WWE, un evento di wrestling, e viene quindi pensata come qualcosa in cui tutti gli sviluppatori si "incontrano" per fare le review del codice.
La Royal Rumble di code Review, come dice il nome stesso, consiste nell'effettuare tutte le code review delle Pull Request in attesa prima dell'inizio di un nuovo task. Se una PR necessita dei cambiamenti, quindi ha dei conflitti o non soddisfa i criteri della Continuous Integration, va in stato *ignore*, salta il round e va direttamente al turno successivo di code review. Inoltre, tutte le PR verranno accettate, e quindi mergiate sul branch principale, alla seconda review approvata da parte di un diverso sviluppatore.
Sintetizzando, le quattro regole principali della RRR sono le seguenti:

1. Prima di iniziare un nuovo task si fanno le review in sospeso.
2. Se una PR ha conflitti o si richiedono cambiamenti dal primo reviewer, va in stato *ignore* finché non sono stati risolti i conflitti o effettuati i cambiamenti.
3. Una PR viene accettata solo dopo la seconda review approvata.
4. Se si sta facendo un task, non si fanno le review.