Manuale del Software Developer

Semplici regole per migliorare le tue abilità da sviluppatore

Carmelo La Gamba

Introduzione

Un buon modo per iniziare questo articolo è sicuramente spiegarne le motivazioni.

Sviluppo software per diletto, studio e infine per lavoro, da ormai oltre dieci anni. Nel tempo ho seguito l'evoluzione di ogni tecnica e metodologia compreso l'avvento di nuovi framework capaci di semplificare tutte le operazioni che fino a qualche anno fa mi impiegavano giorni o settimane.

Al primo anno di università ho affrontato un grosso scoglio: educare la mia mente a ragionare in modo algoritmico. Naturalmente non l'ho fatto da solo ma insieme ad insegnanti e colleghi universitari. Sapevo a cosa andavo incontro durante lo studio di materie vicine alla matematica o fisica ma su tutto ciò che riguardava lo sviluppo del software ero completamente allo sbaraglio. Visto che tutto il percorso universitario era composto per almeno il 60% da materie vicino all'IT posso confermare che è stato come una notte al museo durata cinque anni e due tesi.

Iniziando a lavorare mi sono reso conto di quanto le persone risultino impreparate a dare un contributo di qualità allo sviluppo del codice sorgente. Tutti iniziamo con esperienze in stage o tirocini come semplici programmatori. Soprattutto dopo un percorso universitario è la cosa che si riesce a fare discretamente bene senza bisogno di così tanta formazione aggiuntiva. Troppe volte mi è capitato di sviluppare in team composti da persone eterogenee e soprattutto su progetti partiti male e mantenuti peggio.

Credo molto nella condivisione del codice ragion per cui sviluppo codice open-source ogni volta che ne ho l'occasione, dunque perché non condividere delle regole socialmente valide nell'IT in modo da aiutare la nascita di nuovi ottimi sviluppatori?

Questo articolo **ha il solo scopo** di fornire un indirizzo. Sentitevi liberi di integrare e modificare

Le dieci regole fondamentali

1. **Se non ti piace sviluppare, non lo fare**

   Durante la lettura di un libro, scritto da grandi *software architect*, mi son trovato davanti la seguente frase: «Pensiamo che non abbia senso sviluppare software se non si ha intenzione di farlo bene». Mi ha fatto molto riflettere perché a mio parere si può contestualizzare in ogni ambiente di lavoro. Inutile fare per forza qualcosa se non si riesce a contribuire nel migliore dei modi. Scrivere codice fatto bene è come un operazione chirurgica, tutto il team deve essere compatto e non ci si può permettere di lesionare parti del paziente. Se non è la tua strada proponiti per un altro dipartimento della tua azienda o in altri contesti. Revisionando la celebre frase «Meglio un buon padre che un cattivo prete», con la stessa potenza mistica penso che sia **meglio fare qualcos'altro bene che sviluppare terribilmente**.

2. **Dì «no» al tutto e subito**

   Spesso, nei più svariati contesti, ci sono feature e correzioni da implementare a tempo zero (o addirittura per ieri). Ovviamente le situazioni di emergenza esistono e vanno trattate come tali. Se però l'emergenza diventa la quotidianità le opzioni sono due:

   - Non siete nell'azienda giusta
   - A tutti i livelli di seniority c'è un problema di organizzazione da risolvere

   In ogni caso il mio suggerimento è: **niente panico**. La prima cosa che lo sviluppatore medio pensa è: «ora gli mollo un *taccone* e in mezz'ora sono pronto per il deploy in produzione». Niente di più sbagliato. Agendo secondo questo principio del *tutto e subito* stiamo solamente portando avanti la **teoria dei vetri rotti**.
   La teoria dei vetri rotti sostiene, parafrasando, che un palazzo con una finestra rotta, potrebbe generare fenomeni di emulazione, portando qualcun altro a rompere un lampione o un idrante, dando così inizio a una spirale di degrado urbano e sociale. Ora il nostro palazzo è il codice sorgente e i passanti siamo noi programmatori. Il nostro compito è mantenere il palazzo in ottime condizioni e soprattutto evitare che qualcuno rompa le componenti.

   «Si, bravo, belle parole, ma io ora come faccio? Devo consegnare fra mezz'ora». Nessun problema, l'allenamento è la miglior pratica: più ti allenerai a trovare la soluzione migliore e progettarla con altrettanta cura, più velocizzerai il modo di refattorizzare codice nel più breve tempo possibile. É una skill da non sottovalutare. . . e fa anche curriculum.

3. **Sviluppa codice generico e riusabile**

   Durante la progettazione e lo sviluppo di codice sorgente è naturale prendere come base la richiesta funzionale del cliente per mettere in piedi le prime classi, script, oggetti, funzioni etc.
   Non è sbagliato ma si può incorrere in un errore molto comune cioè la presenza di codice inutilizzabile fuori da quel preciso contesto. Questa situazione porta a duplicazione, perdita di leggibilità e difficoltà nella manutenzione del codice. Tutto questo fortunatamente si può evitare scrivendo quanto più possibile *codice generico*.

   Un esempio pratico è la scrittura di un file Excel, che consiste nella creazione dell'header, nella scrittura del contenuto e l'aggiunta finale di tutte le configurazioni necessarie.
   Si può pensare di creare delle classi che generano file excel a prescindere dal contesto applicativo. Queste classi forniscono dei metodi per ricevere in input l'insieme dei valori da

inserire nell'header e la lista dei valori da scrivere nel contenuto. Come assemblare il file, crearlo e restituirlo (ad esempio sotto forma di *byte[]*) sarà in carico alle classi generiche.
Se in altre parti del software sarà necessario creare un file Excel con altri requisiti funzionali, basterà richiamare la stessa classe e si eviterà parecchio codice duplicato.

Quindi il mio suggerimento è: **quando progetti l'implementazione di una feature, accertati che le componenti siano generiche e riusabili**

4. **Confrontati**

   Il confronto in un team è il vero punto di forza.
   Quando non sai come impostare il codice sorgente chiedi di fare pair-programming ad un collega o al Technical Leader di riferimento: le orchestre funzionano perché sono composte da tanti musicisti con strumenti diversi, non rischiare di fare un assolo di bombardino (non è bello e né orecchiabile, fidati).

5. **Sviluppa codice *ortogonale***

   Questo consiglio è più che attuale ed è la base per le architetture di tipo SOA (Service-oriented architecture).
   L'ideale è progettare e sviluppare componenti software ortogonali cioè completamente isolate. La modifica di ogni componente non deve interferire con il corretto funzionamento delle altre. Immaginate una navicella spaziale diretta verso Marte. A metà del viaggio uno dei pannelli solari smette di produrre energia, gli altri continuano a funzionare indisturbati evitando il collasso immediato di tutto il sistema vitale della navicella spaziale. Il software è come una navicella spaziale, ogni componente è a se stante, ognuna ha il suo scopo e deve avere una **singola responsabilità**.
   Se si progetta software con componenti ortogonali si fa sempre la scelta giusta.

6. **Ragiona e rifletti prima di scrivere codice**

   Durante l'implementazione del codice sorgente, anche quando il tempo a disposizione non manca, tendiamo a non dedicare il tempo necessario per una progettazione adeguata. Magari in quel momento preferiamo optare per una risoluzione veloce e funzionante non badando al concetto di riusabilità e manutenibilità. In poche parole, stai per fare un taccone e nessuno ti potrà fermare. Per evitare di prendere numerosi insulti dai colleghi che malauguratamente si troveranno a dover mantenere quel codice, ti consiglio di porti la seguente domanda: **c'è un'alternativa più valida?** Esiste certamente dunque sviluppala, non vorrei mai che ti mandassero delle macumbe potentissime dove potresti rimanere offeso (semi-cit).

7. **Sviluppa test adeguati** Prima di sviluppare una funzionalità sarebbe meglio sviluppare i test automatici e di integrazione. Tutto ciò non è sempre possibile ma il mio consiglio è di provare sempre ad implementarli. Consapevole del fatto che tantissimo codice sorgente non è coperto da test unitari ci affidiamo alla possibilità che qualche anima di buon cuore possa implementarli anche successivamente: meglio tardi che mai (ma perché non essere proprio noi quell'anima di buon cuore?)
   E per i test utente? Testa le evolutive per almeno 15 minuti. Il numero è del tutto arbitrario, per definire una funzionalità «ben testata» si può scegliere il tempo necessario in base all'entità della *feature*. Penso possa essere utile creare uno schema con tutte le casistiche da testare per ogni funzionalità. L'importante è essere certi di coprire i test utente adeguatamente.

8. **Isolati durante lo sviluppo**

Ad un primo appuntamento è galateo spegnere il telefono per dedicare le giuste attenzioni al partner che si ha di fronte. Mentre si sviluppa codice non bisogna farsi distrarre da colleghi o agenti esterni, bisogna essere un'anima sola con il proprio computer. Il vostro partner è il computer. Procuratevi delle cuffie isolanti o se ne avete l'occasione andate in uffici o posti isolati per almeno due ore . . . e spegnete ogni notifica possibile e immaginabile.

9. **Mantieni aggiornato il team**

   Per portare avanti gli sviluppi ed organizzare il team in maniera semplice spesso si utilizzano strumenti come Board, Kanban o simili. Qualsiasi cosa utilizziate **aggiornatela**. Aggiornare il Team Leader e i colleghi è essenziale. E' come aggiungere dell'olio ad una catena di montaggio e anche una singola persona può inceppare la catena se non olia bene la sua componente.

10. **Leggi libri tecnici**

    Può aiutare tantissimo leggere libri tecnici, guide e manuali per implementare codice o architetture software adeguate. Per iniziare consiglio «The Pragmatic Programmer» di Hunt e Thomas.

Le cinque buone maniere

1. **Rispetta sempre gli standard**

   E' sempre buona norma utilizzare standard per l'impostazione del codice, i design-pattern della Gang of Four sono un ottimo esempio. Potresti guadagnarti la stima e il rispetto dei colleghi se produci software di qualità, al contrario se ti lanci verso un hackathon di spaghetti-code probabilmente finirai per essere il bersaglio durante il gioco a freccette nella sala relax aziendale.

2. **Proteggi il branch principale e segui un versioning ben preciso**

   Non lasciare che il branch *master* sia accessibile da tutti, solo una persona (con una seniority adeguata) deve avere l'onere di eseguire l'allineamento del branch principale. Una volta scelto il pattern di versioning seguilo sempre, evita customizzazioni o di uscire dai binari. Inoltre prima di allineare i branch condivisi (e.g. develop, uat, collaudo, ...) testa il codice in maniera meticolosa.

3. **Commenta e indenta il codice adeguatamente**

   Ad inizio progetto bisogna allineare le configurazioni di indentazione di ogni membro del team. Di solito gli IDE ti consentono di sovrascrivere le configurazioni di default. Sceglietene una e allineate tutti gli IDE dei componenti del team. E' anche un bel gesto commentare (senza scrivere un poema) il codice sorgente e i colleghi apprezzeranno. Creare una documentazione del codice sorgente, sopratutto per le componenti riusabili e generiche, è altrettando un gesto apprezzato. Se segui questo consiglio hai l'opportunità di diventare la rockstar del tuo open-space, non mi farei sfuggire questa occasione.

4. **Coinvolgi subito i colleghi interessati se sospetti un *merge-conflict***

   Se stai sviluppando codice che impatta file sorgente attualmente in lavorazione (o già implementati) di altri componenti del team, coinvolgi subito i colleghi per evitare conflitti successivamente. Oltre ad essere una buona maniera potresti evitare di perdere una numerosa quantità di codice già sviluppato.

5. **Proponi pair-programming in caso di refactoring**

   Se trovi del codice da refattorizzare e vuoi continuare il tuo percorso verso la santità, refattorizzalo. Sarebbe una grande azione anche coinvolgere l'autore del codice da refattorizzare, un buon sano pair-programming migliorerà le skill di programmazione di entrambi.

Cinque consigli aggiuntivi

1. **Non sottovalutare mai i log**

   Proprio cinque minuti fa un mio collega mi ha scritto la seguente frase: «ho un problema in produzione e dai log non riesco a capire nulla, puoi aiutarmi?». Pensate quanto costerà questa *hotfix*: due persone per almeno due ore, per un problema che potrebbe essere risolto da una singola persona nel giro di poco tempo. Un buon sistema di *logging*, seguendo degli standard (in alcuni casi imposti per legge), può salvarti da brutte situazioni più spesso di quanto immagini. Usa dunque tutti i livelli di log (i più usati sono *INFO, DEBUG, WARN, ...*) adeguatamente e ringrazierai il giorno di averli inseriti nel codice.

2. **Ricrea in locale un ambiente simile a quello in cui si effettua il *deploy***

   Ricrea in locale l'ambiente di sviluppo o produzione. Ti eviterà di pronunciare la famigerata frase «a me in locale và», è fastidiossima e ormai non più accettata come giustificazione. Con strumenti come **Docker** è possibile ricreare l'ambiente adeguatamente senza troppi fronzoli.

3. **Leggi il codice degli altri**

   Per migliorare la propria abilità da programmatore si può leggere il codice di altri progetti. Molti grandi progetti di successo oggi sono open-source, vale la pena dare un occhiata per scoprire nuovi modi di implementare codice e di applicare degli standard. E' un pò come leggere un buon libro, alla fine ti rimane qualcosa e contestualmente hai avuto l'occasione di rivivere le nottate di qualche altro collega.

4. **Proponi miglioramenti**

   Proponi qualsiasi cosa, metti in campo le tue idee per migliorare lo sviluppo di un progetto. Si può proporre metodi o tecniche per minimizzare i tempi organizzativi (ad esempio su come diminuire i tempi di uno stand-up meeting usando Scrum) oppure su come scrivere i commenti di un commit. Tutto è utile se proposto per migliorare la vita del team.

5. **Partecipa ai meetup**

   Fare parte di community fuori dal contesto lavorativo può aiutare ad ingrandire il bagaglio culturale e tecnico di un professionista. Informati su gruppi attivi nella tua zona o nel tuo quartiere, partecipa ai meetup organizzati e studia sempre nuove tecnologie che potrebbero interessarti.

Riferimenti bibliografici

[Andrew Hunt, David Thomas] Andrew Hunt e David Thomas (2018). Il Pragmatic Programmer. Guida per manovali del software che voglioni diventare maestri. *Apogeo*.