



PATH TO SENIOR DEVELOPER

Matteo Baccan

Prefazione

Il materiale di questo libro nasce dall'esperienza maturata sul campo, raccogliendo una serie di articoli che ho accuratamente selezionato e sviluppato nel corso di diversi anni di lavoro professionale nel mondo della programmazione.

Gli articoli, originariamente pubblicati su Codemotion Magazine, una delle più autorevoli piattaforme di informazione tecnica per sviluppatori in Italia, rappresentano una testimonianza del mio percorso professionale e delle riflessioni maturate nel corso del tempo.

L'obiettivo è quello di condividere riflessioni, strategie e approfondimenti tecnici che possano rivelarsi preziosi per programmati senior, professionisti che hanno già consolidato una significativa esperienza ma che sono sempre alla ricerca di nuovi spunti di crescita e miglioramento.

Ogni articolo rappresenta un tassello di un percorso di apprendimento continuo, frutto di sfide affrontate, errori commessi e lezioni apprese durante progetti complessi e contesti lavorativi sfidanti.

Attraverso queste pagine, intendo offrire non solo nozioni tecniche, ma soprattutto una prospettiva più ampia sul mestiere del programmatore, condividendo approcci metodologici, best practice e riflessioni che vanno oltre il semplice codice.

Mi auguro che questi contributi, già apprezzati dai lettori di Codemotion Magazine, possano essere un supporto concreto per tutti i professionisti del settore che desiderano ampliare le loro competenze, innovare costantemente il proprio bagaglio tecnico e affrontare con maggiore consapevolezza e maestria le sfide sempre più complesse del mondo dello sviluppo software.

E se leggendo alcuni capitoli troverai idee che vanno in contrasto con quello che

pensi o se vorrai aggiungere qualcosa, sentiti libero di scrivermi. Questo libro verrà costantemente aggiornato e arricchito grazie ai tuoi feedback.

DRAFT DRAFT DRAFT

Ringraziamento

Grazie alla mia famiglia, che con il suo amore e il suo supporto incondizionato mi ha permesso di realizzare questo progetto. Senza la sua pazienza, comprensione e incoraggiamento, questo traguardo non sarebbe stato possibile.

Un ringraziamento speciale va anche agli amici che mi hanno sostenuto lungo il percorso, offrendo consigli preziosi, critiche costruttive e momenti di condivisione che hanno arricchito il mio lavoro.

Infine, voglio esprimere la mia gratitudine a tutti coloro che, direttamente o indirettamente, hanno contribuito alla realizzazione di questo libro. Ogni parola, ogni pagina, è frutto di un lavoro collettivo e di una passione condivisa.

"La gratitudine non è solo la memoria del cuore, ma anche la luce che illumina il cammino futuro." - Anonimo

Grazie di cuore.

Introduzione

Nel vorticoso universo della tecnologia, dove il codice è il nuovo linguaggio universale, diventare un programmatore senior non è semplicemente una questione di anni di esperienza, ma di crescita continua, passione e strategia. Questo libro nasce dall'obiettivo di trasformare programmati volenterosi in professionisti esperti, capaci di affrontare le sfide più complesse del mondo dello sviluppo software.

Ogni capitolo è stato accuratamente progettato come una tappa di un viaggio di trasformazione professionale. Non troverai solo nozioni tecniche, ma una roadmap completa che abbraccia competenze tecniche, soft skill, metodologie di lavoro e strategie di apprendimento. Dal miglioramento delle tue capacità di programmazione all'acquisizione di una mentalità da senior developer, questo libro ti guiderà attraverso un percorso di crescita professionale unico.

Che tu sia un giovane programmatore con pochi anni di esperienza o un professionista che cerca di raggiungere il successivo livello di eccellenza, queste pagine sono pensate per te. Sono il risultato di anni di esperienza sul campo, di successi, di fallimenti e apprendimenti continui nel mondo dello sviluppo software.

Preparati a un viaggio che andrà oltre il semplice scrivere codice: imparerai a pensare come un vero professionista, a risolvere problemi complessi e a costruire la tua carriera con consapevolezza e strategia.

"Diventa programmatore dopo i 40 anni"



Chi, come me, passa molte ore online, non può non aver mai visto questo titolo associato alla vendita di corsi di programmazione.

Si tratta di un titolo al limite del clickbait, che cerca di convincere le persone che, anche dopo i quarantanni si può diventare facilmente dei programmatori.

Lo scopo è chiaro: visto che la carriera di un programmatore inizia normalmente molto presto, si vuole alzare l'asticella prendendo un bacino di persone che ormai credevano di non potercela più fare o che magari sono nella condizione di non avere un lavoro che li soddisfa e vogliono fare altro: cercano la classica ricollocazione professionale.

La realtà

La ragione di questa tendenza risiede nella crescente domanda di figure professionali nel settore della programmazione, dove attualmente è difficile trovare individui con le competenze necessarie per soddisfare la richiesta del mercato.

Anche se è lodevole che si cerchi di insegnare la programmazione anche a persone che ne sono a digiuno, non condivido il modo col quale si vuol far passare questo concetto.

Sia chiaro, sono sempre a favore dei corsi di programmazione e vedo con benevolenza chi vuole impegnarsi e cambiare una situazione che non lo soddisfa, ma sono contro ai messaggi che inducono a pensare che qualcosa che richiede tempo e dedizione diventi improvvisamente alla portata di tutti.

Quella frase nella mia testa suona come un "Diventa un chirurgo dopo i 40 anni" o "Diventa un avvocato dopo i 40 anni". Non è che non sia possibile, ma è difficile, molto più difficile che a venti, soprattutto se nella vita si è fatto tutt'altro.

Ritengo che ogni individuo abbia il diritto di scegliere il proprio percorso nella vita: ci si può svegliare un giorno e

decidere che la programmazione è il proprio futuro, così come un programmatore può decidere di diventare un allevatore di mucche.

Proviamo però a portare un po' di realtà in questa frase e ad immaginare cosa accadrà alla fine del corso che ti trasformerà in un programmatore, indipendentemente dalla qualità dell'insegnamento e dalle capacità personali.

Inevitabilmente, occorrerà cercare un posto di lavoro dato che tentare la fortuna del freelance senza contatti ed esperienza potrebbe essere una prospettiva piuttosto frustrante.

Una volta trovata un'azienda che è disposta ad investire su di te, ci potrebbero essere una serie di bias che potrebbero cambiare il modo in cui i potenziali datori di lavoro si avvicinano a te. Il primo di questi bias è sicuramente l'età.

Bias anagrafici

Proviamo a considerare due scenari per semplificare il ragionamento:

Se hai 20 anni e ti addentri nel mondo della programmazione, dopo aver fatto un corso o un percorso universitario, le aspettative su di te saranno indulgenti. La mancanza di esperienza viene normalmente perdonata e ci si aspetta una grande voglia di apprendere. Se l'azienda crede veramente nella formazione, non è inusuale avere anche un mentore disposto ad aiutarti nel tuo percorso di crescita professionale.

D'altra parte, se hai 40 anni, le aspettative nei tuoi confronti sono maggiori e ci potrebbero essere molti bias che vengono messi prima della tua preparazione. Bias anagrafici, bias legati alla tua situazione familiare, bias sui tuoi limiti di apprendimento rispetto ai ragazzi di vent'anni: veloci a programmare e con molto più tempo a disposizione.

Queste discrepanze di aspettative rappresentano un aspetto cruciale da considerare e per il quale è necessario prepararsi mentalmente.

Occorre essere molto motivati se si vuole percorrere questa strada che, non è impossibile, ma all'inizio può essere veramente dura e potrebbe sviluppare una profonda frustrazione. È importante tenere presente questi aspetti e non farsi scoraggiare dalle difficoltà iniziali, perché solo col tempo si riusciranno a dimostrare le proprie capacità e a far valere la propria esperienza.

In definitiva, non esiste una regola fissa, ma è fondamentale essere consapevoli di queste dinamiche per evitare di sottovalutare le sfide che potrebbero presentarsi dopo il completamento del corso e durante la ricerca di un impiego nel settore della programmazione.

Sicuramente uno dei fattori mitiganti, che può abbattere o comunque mitigare qualsiasi preconcetto è quello di dimostrare di essere fortemente motivati, non solo a parole, ma anche con azioni tangibili. Mostrarsi attivi su piattaforme come GitHub, partecipare ad eventi del settore e collaborare a progetti open-source sono tutti elementi che dimostrano un impegno concreto e una passione per la programmazione.

Questi "piccoli tasselli" non solo testimoniano la volontà di apprendere e di mettersi in gioco, ma possono anche rivelare un talento sopito che inizia ad esplodere in quel momento, contribuendo a rafforzare la credibilità e l'attrattiva agli occhi dei potenziali datori di lavoro.

La proposta formativa

Vorrei ora soffermarmi anche su un altro aspetto che emerge leggendo con più attenzione alcune proposte formative, ho trovato qualcosa che mi ha lasciato perplesso e che mi ha spinto a scrivere questo articolo.

Un commento ricorrente, che si riscontra in varie forme sotto questo tipo di

percorsi e che vorrei riportare come esempio, è il seguente:

"Noi insegniamo come diventare un architetto software, in modo costante ... per non dover fare la gavetta".

La partenza di questi corsi è spesso focalizzata su "programmatori quarantenni" e altrettanto spesso prova a vendere il concetto che le persone diventeranno "architetti software senza gavetta".

Posso comprendere il desiderio di offrire corsi a persone che aspirano a diventare programmatori, e riconosco l'esistenza di un mercato in cui cercare potenziali talenti. Tuttavia, rimango perplesso di fronte al messaggio che si vuole trasmettere, secondo cui diventare architetti software dopo un corso non richiederebbe "gavetta".

È importante sottolineare che l'esperienza, o "gavetta", è fondamentale per sviluppare una comprensione accurata della progettazione di software funzionale, in grado di soddisfare le esigenze dei clienti e degli utenti.

La "gavetta" implica commettere errori, imparare da essi e affrontare sfide pratiche in diversi progetti. Significa prendere decisioni difficili, talvolta in contrasto con le best practice, perché il contesto o le esigenze del progetto lo richiedono.

Tagliare corto sulla "gavetta" rischia di produrre progetti che, sebbene possano sembrare ben realizzati, non riescono a tradursi in soluzioni praticabili a causa di problemi quali costi, prestazioni, tempi o semplice mancanza di esperienza.

Se dopo aver completato un corso e ottenuto il titolo di "architetto software", ci si chiede perché non si riesce a trovare lavoro, la risposta è semplicemente la mancanza di "gavetta". Nessun corso può sostituire l'esperienza pratica e l'apprendimento che essa comporta: la sola partecipazione ad un corso non fa di voi un programmatore professionista o un architetto software.

Come affermato da chi ha pestato la tastiera più volte di me: "Non esiste una scorciatoia per diventare un programmatore professionista. È necessario affrontare sfide reali e imparare dagli errori lungo il percorso." ("The Pragmatic Programmer: Your Journey to Mastery"). Questa frase cattura in modo eccellente il concetto che la programmazione è un'abilità che può essere insegnata, ma è solo attraverso la pratica che si può aspirare a diventare dei veri professionisti.

Per alzare il livello, quando si parla della figura dell'architetto software, ci troviamo di fronte a un discorso diverso.

Essere un architetto software non è qualcosa che si improvvisa; richiede anni di esperienza e pratica. Si tratta di un lavoro che coinvolge l'intera struttura di un'azienda e ne può decretare il fallimento o il successo in base al modo col quale viene affrontato il lavoro.

Anche se un corso di formazione può essere utile e approfondito, non può sostituire l'esperienza maturata nel tempo e fornirà sempre una visione parziale del lavoro di un architetto software.

Un altro aspetto importante è quello di non confondere le figure di "programmatore" e "architetto software": sono due ruoli distinti, ognuno con le proprie competenze e esperienze. Essere un bravo programmatore non implica necessariamente essere un bravo architetto software e viceversa. Tuttavia, è innegabile che un background da programmatore possa essere vantaggioso per aspirare a diventare un buon architetto software.

Possiamo fare un parallelo con il mondo del calcio: molti allenatori di successo hanno avuto una carriera da calciatori e hanno vissuto in prima persona le dinamiche del gioco. Allo stesso tempo, ci sono casi di giocatori che, spinti dalla fretta di diventare allenatori, hanno bruciato le tappe senza avere la giusta esperienza, mettendosi in gioco in squadre importanti e compromettendo la loro credibilità e carriera.

Mentre la pratica è fondamentale per diventare dei bravi programmati, diventare un architetto software richiede una profonda esperienza e una comprensione del settore che solo il tempo può fornire. Bisogna quindi essere consapevoli dei percorsi e dei tempi necessari per raggiungere gli obiettivi professionali desiderati.

Conclusioni

Chiunque può diventare un programmatore, a qualsiasi età, ma è importante essere consapevoli delle sfide e delle aspettative che si presenteranno lungo il percorso.

Più inizierete tardi, più sarà difficile, ma non impossibile. La passione e la dedizione possono aiutare a superare le difficoltà iniziali e a dimostrare il proprio valore nel settore della programmazione.

Dopo alcuni anni di esperienza professionale, capirete quanto sia fondamentale affrontare la fase di apprendimento pratico e quanto sia illusorio credere che un corso possa sostituire l'importanza della "gavetta" nel settore della programmazione.

La pratica e l'esperienza sul campo sono pilastri irrinunciabili per una carriera di successo nella programmazione. Sebbene i corsi possano fornire una base teorica solida, è solo attraverso il lavoro pratico e la continua ricerca che si può realmente acquisire la competenza e la padronanza necessarie. Pertanto, è essenziale riconoscere l'importanza della "gavetta" e mantenere aspettative realistiche riguardo ai percorsi di formazione e sviluppo professionale nel settore della programmazione.

"Ho fatto il corso, ma se non lo metto in pratica lo dimentico": diventerà una frase che sentirete spesso e che vi farà riflettere sulle reali competenze acquisite.

Riferimenti

[The Pragmatic Programmer: Your Journey to Mastery](#)
[Diventare programmatore ai 40 anni: è possibile?](#)

**I programmati che concludono tutti i task non hanno finito la
loro giornata lavorativa**



Sta emergendo tra i programmatori e i team di sviluppo un approccio troppo individualista e dannoso nel modo in cui vengono gestiti i progetti software. L'obiettivo delle persone sembra diventato solo quello di chiudere i propri task il prima possibile, senza preoccuparsi di come questo possa impattare il lavoro degli altri membri del team e del fatto che questo aspetto non porta che un valore effimero al progetto.

Chiudere i task non è l'obiettivo finale

Questo approccio potrebbe anche essere il primo segnale di un mancato governo da parte dell'azienda, si verifica nei progetti fortemente strutturati a task, dove ad ogni persona viene assegnato un compito che deve essere completato il prima possibile e su questo compito viene poi valutata la performance del proprio lavoro.

Quel programmatore è una macchina da guerra, finisce tutto quello che gli viene assegnato già al mattino e poi si mette a fare altro

Quante volte avete sentito questa frase e il compiacimento diffuso da parte di colleghi e management? Ma è davvero così positivo?

Per riuscire ad avere certe performance, si tende a chiudere la persona in un ambiente isolato, asciugarla da ogni preoccupazione, per spingerla al risultato massimo.

Da un certo punto di vista, la cosa è positiva: vengono tagliate tutte le distrazioni, le continue riunioni, le interminabili telefonate e le situazioni di multitasking in modo da focalizzarsi sull'obiettivo. Da un altro punto di vista, si perde di vista il quadro generale e si rischia di creare un ambiente di lavoro tossico, fortemente individualista, in cui le persone non si aiutano e non condividono le proprie conoscenze.

Il fatto che poi molti programmati siano persone propense all'isolamento quando svolgono il proprio lavoro non giova a questa situazione.

La sindrome del cavallo da corsa

Capite quanto sia facile, in un ambiente come questo, non accorgersi dei malumori o dello stacco fra le varie figure, se si guarda solamente il risultato nel breve termine. Come un cavallo da corsa indossa i paraocchi per correre veloce verso al traguardo senza guardare il mondo esterno, così i programmati sono lanciati nella risoluzione dei propri task.

L'importante è la fine del task, non l'economia del progetto

Premiare l'individualismo

Viene quindi premiato l'individualismo, legato al risultato tangibile.

Peccato che un progetto è composto anche da una serie di risultati intangibili, come la condivisione delle conoscenze, la collaborazione e la crescita personale e professionale delle persone coinvolte.

Le metriche dettate dal management e da molti tool che vengono utilizzati in azienda tendono a valutare tutto quello che è tangibile: il numero di task chiusi, le linee di codice scritte, il tempo fra l'apertura di un problema e la sua risoluzione: difficilmente tengono conto dei task intangibili.

Pensate a tutte le volte che avete dato un aiuto a un collega per sbloccarsi da un problema, alla correzione di una riga di codice che risolve un problema noto da mesi e mai risolto, alle sessioni di pair programming o alla dritta detta davanti al caffè.

Occorre quindi pensare ai progetti in modo olistico e cercare di staccarsi dalla logica del task come unico obiettivo. Un team lavora insieme deve raggiungere un

obiettivo comune. Non vincono i singoli, ma vince il gruppo che è in grado di evolvere e migliorare il prodotto in modo continuo e armonico.

Il problema del "finire i task"

La vera sfida è quella di riuscire a trasformare una serie di persone, inserite in un contesto dove viene premiata la performance del singolo, in una squadra che pensa, agisce e lavora ad obiettivi nel lungo termine.

Fare squadra non è solo una questione di soft skill, ma è anche una questione di cultura aziendale e di come vengono valutate le persone all'interno dell'azienda.

Creare compartimenti stagni non giova a nessuno e deve essere bilanciato con delle attività in grado di portare del valore anche verso le persone più lente o isolate, in modo da poterle trainare ad un miglioramento culturale e personale.

Quando si hanno team composti da persone più o meno esperte, se si riesce a trovare il modo per versare delle conoscenze da una persona all'altra, diventa un arricchimento sia per il progetto che per le singole persone.

Per chi gestisce un progetto software, è fondamentale utilizzare al meglio il tempo delle persone in modo da accrescere il valore del progetto oltre alla chiusura dei task.

È essenziale favorire una cultura collaborativa all'interno del team, in cui le persone possano aiutarsi reciprocamente e condividere conoscenze.

Di chi è la responsabilità?

Sicuramente favorire un approccio collaborativo è responsabilità del management, ma anche i singoli membri del team hanno un ruolo importante in questo processo. È fondamentale che i membri del progetto siano consapevoli del fatto che il loro lavoro non è solo quello di chiudere i propri task ed andare a casa

il prima possibile.

Nelle riunioni periodiche, negli stand-up, deve essere compito dei membri del team informare gli altri se ci sono dei blocchi o se hanno bisogno di aiuto. Allo stesso modo, i membri del team devono essere pronti ad aiutare i colleghi condividendo le proprie conoscenze e competenze.

Non è però detto che questo aspetto emerga in modo spontaneo. Per questo è necessario che il management promuova attivamente la collaborazione, cogliendo queste situazioni e spronando le persone a lavorare in modo collaborativo.



Quali strategie possiamo mettere in atto per mitigare questo problema?

Esistono vari modi per evitare che questo modo possa compromettere l'economia del progetto.

Se parliamo di soft skill, dobbiamo sicuramente inserire nel gruppo di lavoro persone che abbiano una predisposizione naturale al lavoro di squadra, che sappiano lavorare in gruppo e che siano in grado di condividere le proprie conoscenze.

Se una persona finisce i propri task e non ha nulla da fare, potrebbe essere spronata a fare code review del lavoro degli altri membri del team, a fare refactoring del codice appena aggiunto o a fare mentoring e coaching delle persone meno esperte, e su questo tipo di attività dovrebbe essere valutata la performance del singolo.

Gestione delle superstar

In un mondo perfetto, tutti riescono a fare il proprio lavoro, pranzano a casa con la famiglia e al pomeriggio giocano a tennis.

In un mondo normale ci si trova di fronte a dei problemi e non è detto che tutti siano risolvibili con le proprie forze. A volte è necessario farsi aiutare da i cosiddetti "programmatori superstar".

A tal proposito mi viene in mente una discorso di Ottavio Bianchi ai calciatori del Napoli quando arrivò Maradona in squadra:

Lui è Maradona e noi siamo noi. Il problema nasce se qualcuno di noi vuole fare il Maradona. Se accettiamo questo ovvero che lui è lui e noi siamo un'altra cosa possiamo vincere, ma se noi noi non lo accettiamo o qualcuno di noi non lo accetta la cosa non funziona.

A volte la superstar è necessaria per chiudere alcuni task: immaginate tutte le situazioni nelle quali vi siete trovati di fronte a un problema che una persona

molto competente sull'argomento avrebbe risolto in pochi minuti, mentre voi ci avete impiegato giorni.

In questi casi è facile che la singola persona lavori su task ben definiti e che non abbia, o abbia scarso bisogno di aiuto da parte degli altri.

L'ossessione al completamento del task da parte della superstar non è un problema, ma è fondamentale che il management sia in grado di gestire queste situazioni e di garantire che il lavoro della "superstar" sia comprensibile e mantenibile da tutti i membri del team.

Per questo è fondamentale che il team analizzi successivamente il codice scritto dalla "superstar", lo riveda, lo documenti e ne faccia refactoring, per poterlo rendere comprensibile e mantenibile da tutti.

L'effetto "catena di montaggio"

Dobbiamo quindi evitare di pensare alla programmazione come a una catena di montaggio, in cui ogni persona è responsabile di un singolo task e non ha alcuna responsabilità nei confronti degli altri.

Perché hai messo questa riga di codice che rallenta il programma?

Perché in questo modo ho risolto una segnalazione di sicurezza.

Ma non ti sei accorto che potevi fare in questo modo?

No, non mi sono accorto, non era il mio task.

Attenzione però a non abusare dell'approccio inverso dove, appena una persona finisce le proprie attività, viene assalita con l'assegnazione continua di nuovi compiti o chiusura dei lavori di altre persone.

Questo potrebbe portare a situazioni in cui le persone percepiscono un aumento dello stress e un carico di lavoro eccessivo, che li porta a non gestire nel modo ottimale i propri compiti.

Il "burnout" è un problema reale e tangibile e una situazione mal gestita potrebbe favorire l'insoddisfazione lavorativa e l'allontanamento delle persone migliori, che si sentono sopraffatte dal numero di attività che devono portare a termine.

Allo stesso modo ci potrebbero essere persone che rallentano volutamente il proprio lavoro, per evitare di dover fare il lavoro degli altri. In questi casi diventa fondamentale che il management sia in grado di gestire queste situazioni, riesca a garantire che il carico di lavoro sia equamente distribuito tra i membri del team e metta in atto una serie di strategie per migliorare la collaborazione all'interno del team.

Conclusioni

Un buon programmatore non è colui che finisce tutti i task assegnati, ma colui che viene messo in grado di lavorare in modo collaborativo e di condividere le proprie conoscenze con gli altri membri del team.

Occorre saper far crescere il gruppo di lavoro nel quale si è inseriti, ed è importante che il management non valuti i propri risultati solo da dati numerici, ma sia in grado di gestire le persone, ascoltarle, spronarle e capire dove sono le differenze per appianarle.

Il valore che può dare un senior va oltre la risoluzione dei task, deve estendersi alle attività di coaching e mentoring, in modo da far crescere chi ha meno esperienza.

Questo è il modo migliore per garantire che il progetto software sia di successo e che il team sia in grado di crescere e migliorare nel tempo.

La forza di ammettere di non sapere



Il mondo della programmazione è un universo in continua evoluzione, dove le competenze tecniche rappresentano solo il punto di partenza.

Tutti guardano con timore il mondo delle AI, sulla carta software in grado di conoscere più di qualsiasi programmatore e capaci di generare codice con la facilità di bere un sorso d'acqua, ma la differenza tra un buon programmatore e un grande programmatore non risiede solo nella capacità di scrivere righe di codice, ma anche nell'abilità di interpretare correttamente le specifiche, di vedere oltre quanto viene richiesto, di riconoscere quando una strada è sbagliata e sarebbe opportuno intraprenderne un'altra.

Tutte queste capacità si acquisiscono con l'esperienza e al momento non esistono AI con tali caratteristiche.

Se ti definisci programmatore, ma hai comunque paura: è normale.

La paura di non sapere abbastanza, di essere superato da un software è un'ombra che accompagna costantemente il percorso professionale di ciascun programmatore.

Se hai appena capito la differenza tra un `for` e un `while` e ti scende una goccia di sudore quando vedi una shell: è perfettamente normale. Sei un junior e ti spaventi per cose che un giorno imparerai a fare con gli occhi chiusi.

Ma questa paura grava anche sulle spalle di chi lavora da anni, che ha scritto più cicli for che il proprio cognome, che identifica i bug solo guardando il codice e che possiede una visione olistica dei progetti.

La paura di non sapere

La paura di non sapere è una costante di questo lavoro che deve essere presto accettata e superata, poiché ci blocca, impedendoci di crescere e migliorare.

Per fare il programmatore servono competenze, e più se ne acquisiscono, più ci si rende conto che ce ne sono sempre di nuove da apprendere. Allo stesso tempo non è possibile fermarsi, limitarsi a una nicchia o specializzarsi in un singolo ambito, perché ci sarà sempre qualche sviluppatore che, magari lavorando dalla sua capanna in cima a un albero, creerà codice destinato a diventare fondamentale per il tuo lavoro.

Non possiamo conoscere a pieno tutti i prodotti che utilizziamo né tutto ciò che viene rilasciato ogni giorno: o dedichiamo tempo a capire cosa succede o passiamo le giornate a scrivere codice. Ma anche se fossimo in grado di fare entrambe le cose, non saremmo comunque in grado di conoscere tutto.

Chi sa tutto? Le AI? No, nemmeno loro. Apprendono quanto riescono ad assimilare dalla rete, ma se il dato non è presente o viene interpretato male, il risultato sarà errato: esiste un mondo di codice in ambito legacy, privo di documentazione che le AI non conoscono. Ci sono tutti i prodotti closed source che non possono essere analizzati, tutti quelli che devono ancora essere rilasciati e così via.

Quindi mettiamoci l'anima in pace: non possiamo sapere tutto e anche con gli strumenti più moderni non saremo mai in grado di conoscere ogni cosa. Ci saranno sempre gradi di ignoranza o di errore in tutto il codice che andremo a scrivere.

Allo stesso modo, tutto quello che scriviamo oggi potrebbe rivelarsi sbagliato: perché non conosciamo a pieno un prodotto e perché la tecnologia evolve e cambia, e ciò che oggi è considerato corretto, domani potrebbe essere ritenuto errato.

Ero ignorante e non lo sapevo

Ultimamente ho lavorato sull'ottimizzazione di un prodotto Java al quale veniva chiesto di eseguire una serie di operazioni entro la soglia di 10 millisecondi.

Il codice sembrava corretto, ma saltuariamente il programma superava i 100 millisecondi.

In un ambiente dove anche i millisecondi avevano un peso, questa oscillazione non era tollerata perché superava lo SLA del servizio che dovevamo fornire.

Qual era il problema? L'istanziazione di un oggetto richiamava il classloader per cercare una certa tipologia di classe, invece di istanziare in modo diretto una classe. Questa operazione, che nel 99,999% dei casi si risolveva in meno di 1 millisecondo, in alcuni casi arrivava a 100 millisecondi.

La soluzione? Abbiamo evitato la ricerca della classe in modo dinamico, passando a un'istanziazione statica.

Peso della modifica: 1 riga di codice, ma il ritardo saltuario era sparito.

Non conoscevamo così in profondità la classe che usavamo, e in condizioni normali il problema non sussisteva: appariva solamente in momenti di stress applicativo in forte concorrenza.

Come l'abbiamo capito? Parlandoci fra di noi, confrontandoci, mettendo insieme le nostre competenze ed esperienze.

L'ignoranza è una risorsa

In questo mare di conoscenza – o di ignoranza, se si vuole vedere il bicchiere mezzo vuoto – l'unica cosa che possiamo fare è potenziare quelle che sono le soft skill, un tempo sottovalutate e oggi più che mai essenziali.

Sia ben chiaro: le competenze tecniche sono fondamentali, così come è essenziale conoscere e usare bene i nuovi strumenti che ci vengono messi a disposizione ogni giorno, ma per chi ha trascorso anni a lavorare con il codice, ci sono delle skill che fanno la differenza quando si collabora in team.

Assodato che libri, manuali e AI non avranno mai tutte le risposte, la cosa che apprezzo maggiormente è il **coraggio di chiedere**.

Ci sono competenze ed esperienze che risiedono solo all'interno delle persone, e spesso è più facile chiedere a chi sa piuttosto che cercare di capire da soli.

Per paura di essere umiliati o derisi, non osiamo fare una domanda a un collega, non diciamo "non ho capito", non abbiamo il coraggio di chiedere chiarimenti. Passiamo ore davanti a pezzi di codice incomprensibili, chiediamo aiuto in forma anonima su forum, Reddit, Stack Overflow o a varie AI, ma piuttosto che rivolgere una semplice domanda al collega seduto accanto a noi, preferiremmo quasi darci fuoco.

Non facciamo gruppo e preferiamo spaccarci la testa da soli di fronte a un problema.

L'orgoglio si trasforma in una catena invisibile che ci limita, ci paralizza, ci fa sprecare tempo prezioso e ci induce a commettere errori evitabili.

Temiamo di essere giudicati come deboli, ma in realtà, ammettere di non sapere è una grande virtù.

Una volta un antico filosofo disse:

Il tuo mantra per la vita deve essere "dignità zero"

In realtà non era un filosofo, ma Mauro Repetto, fondatore degli 883, ma il messaggio che arriva è lo stesso: **seguire i propri sogni senza preoccuparsi delle critiche o del giudizio altrui**.

Molte volte è infatti la paura del giudizio che ci ferma e non ci permette di crescere. Ma se non chiediamo, come possiamo imparare? Se non ammettiamo di non sapere, come possiamo migliorare?

Gestione dell'ignoranza

Siamo tutti ignoranti, ma in argomenti diversi. Ammettere di non sapere e palesare questo aspetto in azienda a volte può far muovere delle risorse.

Anni fa mi trovavo in un'azienda dove avevamo deciso di puntare su C++ per il nuovo prodotto che volevamo realizzare: i motivi erano la velocità di esecuzione e la possibilità di scrivere codice più vicino all'hardware.

Col tempo la scelta si è rivelata sbagliata, non per i presupposti iniziali, ma per il fatto che si rivolgeva a un bacino di programmatore che non avevano mai lavorato con quel linguaggio e provenivano da linguaggi più semplici.

In quel frangente il management si accorse di una lacuna: il team nel quale ero inserito e che doveva sviluppare in C++ non aveva, o aveva solo in parte, adeguate competenze per poter realizzare il prodotto che ci eravamo prefissati.

Per questo motivo venne organizzato un corso con un esperto del linguaggio, per sbloccarci su tutta una serie di aspetti che non avevamo considerato e non conoscevamo.

Palesare la nostra ignoranza ci fece fare un salto in avanti, ci permise di capire dove stavamo sbagliando e di correggere il tiro, ma soprattutto ci fece comprendere che non siamo soli e che chiedere aiuto è normale.

Se questo vale per un team, dove forse è anche più facile chiedere, vale molto anche per i singoli.

Il coraggio di chiedere

Anni fa lavoravo per un'azienda che aveva assunto diversi ragazzi freschi di diploma. Le loro competenze erano variegate: alcuni dimostravano già un certo talento, altri meno. Tuttavia, uno in particolare si distingueva dagli altri non per le

sue capacità tecniche - anzi, aveva notevoli lacune - ma per la sua straordinaria predisposizione a chiedere chiarimenti, anche ripetutamente durante la giornata. Questo atteggiamento, apparentemente semplice ma in realtà rivoluzionario, con il passare del tempo lo ha portato a superare tutti i suoi coetanei.

Il suo ChatGPT erano i colleghi, le persone più esperte di lui, che lo indirizzavano, gli spiegavano cosa occorreva fare e non fare e dove doveva migliorare il proprio studio.

La paura di fare brutte figure è spesso una gabbia che ci tiene intrappolati nella nostra comfort zone. Abbiamo una scelta: rimanere prigionieri della paura o liberarci con il coraggio di chiedere. La strada per diventare programmati migliori inizia da qui.

Il mito del tuttologo

Il mondo tech abbonda di persone che si ergono a tuttologi, convinte di sapere tutto e di non aver bisogno di alcun aiuto esterno. Questa pericolosa illusione nasce da un'immagine distorta della perfezione.

Considerate i supereroi come metafora: vengono rappresentati come esseri perfetti, infallibili, autosufficienti e impavidi. Ma noi non siamo supereroi con superpoteri: siamo esseri umani, con tutti i nostri limiti e le nostre fragilità.

Provate a guardare anche tutte le persone che seguite sui social. Spesso partono da argomenti che conoscono, sono molto spigliate nel farlo e grazie a questo attirano follower.

Col tempo, in modo più o meno inevitabile, la vena creativa diminuisce e la ricerca spasmodica di argomenti interessanti porta a parlare di cose che non si conoscono bene, ma che si pensa possano interessare.

A questo punto la qualità si deteriora, ma visto che il pubblico continua ad apprezzare e i follower aumentano, si persevera su questa strada, fino a quando

non ci si ritrova a parlare di argomenti che non si conoscono affatto.

Questo è il momento in cui si diventa tuttologi, si crede di poter parlare di tutto e di sapere tutto, ma in realtà si è solo un bluff.

Questa situazione capita più o meno a tutti col tempo, ma è importante riconoscerlo e non cadere nella trappola del tuttologo.

Se il tuo seguito ti vede come una guida, come un punto di riferimento, è importante a un certo punto fermarsi, capire il proprio limite, ammettere di non sapere e far intervenire chi è esperto di un certo argomento.

La differenza fra un tuttologo e un esperto è tutta qui: se siamo esperti e vogliamo il meglio per un certo argomento, andremo a chiedere a chi, su un particolare tema è più ferrato di noi e saprà meglio indirizzarci, e non sempre si tratta di una AI, ma di una persona fisica, con un nome e cognome.

Chiedere da solo però non basta

Abbiamo trovato il coraggio di chiedere? Di far intervenire l'esperto? A questo punto potrebbe nascere un secondo problema: fingere di capire.

Già chiedere è un grande passo, ma ammettere di non aver capito una risposta è un secondo passo che spesso non si fa. Si preferisce fare finta di aver compreso, anche se non è così, per paura di essere giudicati come stupidi.

Repeat and rephrase

Un modo semplice per verificare la comprensione di un concetto è ripeterlo e riformularlo con le proprie parole. Se non si riesce a farlo, significa che non si è capito bene e occorre chiedere ulteriori chiarimenti.

La differenza fra prosciutto cotto e prosciutto crudo

Io non riesco a distinguere il prosciutto cotto dal prosciutto crudo, è qualcosa che mi porto avanti da un sacco di anni. Il mio cervello si rifiuta di associare la parola al prodotto. So cosa voglio, ma non riesco ad associare la giusta parola.

Passami il prosciutto cotto

è una delle domande più imbarazzanti che mi possano fare, perché so che la risposta sarà:

Quello è il prosciutto crudo

Col tempo ho capito il problema: assocavo al colore scuro la cottura e al colore chiaro la non cottura dell'alimento. Il mio cervello assocava al contrario le parole, e invertivo il risultato, sapendo che c'era qualcosa che non andava, ma non riuscendo a capire cosa.

A furia di sbagliare ho iniziato a chiedere, e dopo tante spiegazioni, tanti modi per ricordare, ho capito come distinguere il prosciutto cotto dal prosciutto crudo: ma se non avessi iniziato a chiedere, forse non avrei mai avuto un meccanismo in testa in grado di farmi capire la differenza.

La paura del giudizio

Essere giudicati, essere visti come quello che non sa, è una paura che ci blocca e ci impedisce di crescere. Ma se non chiediamo, come possiamo imparare?

Allo stesso modo, se superiamo questa paura nella domanda, dobbiamo superarla anche nella risposta: non basta chiedere, occorre anche capire la risposta e ammettere di non comprendere.

Chiedere senza capire equivale a non chiedere, ma anche ammettere di non aver compreso la risposta è un'arte che deve essere appresa col tempo.

Fail-forward culture

Un altro concetto che mi affascina è quello della fail-forward culture, ovvero la cultura del fallimento. Fallire è normale, è umano, è un passo necessario per crescere e migliorare.

Ammettere di non sapere, sbagliare e usare il fallimento come trampolino di lancio per migliorare è una delle chiavi per diventare un programmatore migliore.

Purtroppo non sapere, fallire, mostrarsi vulnerabili viene ancora percepito come un difetto: in realtà rappresenta un potente strumento per crescere, imparare e migliorare, una lezione che dovrebbe essere trasmessa fin dall'infanzia.

Non trattenerti per paura di sbagliare, ma sbaglia per imparare

Conclusioni

La prossima volta che esitate a fare una domanda, chiedetevi: preferisco apparire momentaneamente insicuro o rimanere bloccato per sempre?

E se qualcuno dovesse deridervi per le vostre domande apparentemente ingenue, ricordate che è proprio lui, non facendole, a limitarsi e a costruire lentamente una gabbia invisibile dalla quale sarà sempre più difficile uscire.

Nessuno nasce sapendo tutto, ma solo chi ha il coraggio di ammettere di non sapere può veramente imparare e crescere.

Oltre il Full Stack: Percorsi di Crescita nello Sviluppo Software

Il viaggio per diventare uno sviluppatore senior è un processo complesso che va ben oltre la semplice acquisizione di competenze tecniche. "Oltre il Full Stack: Percorsi di Crescita nello Sviluppo Software" esplora quattro pilastri fondamentali di questa trasformazione professionale, partendo dalla decostruzione del mito del Full Stack Developer - descritto come una figura che "non è un esperto di backend, né di frontend, né di DevOps; conosce le nozioni di UX, UI, SEO, marketing, sa un po' di tutto, ma nulla in modo approfondito" - fino all'importanza delle code review e della manutenzione del codice.

Questi capitoli ci guidano attraverso una metamorfosi professionale essenziale: dal superamento dell'illusione di dover padroneggiare ogni aspetto dello sviluppo software alla consapevolezza delle conseguenze di ogni modifica al codice ("ogni modifica, anche la più banale, può comportare problemi prestazionali, di regressione, di sicurezza, di manutenibilità").

Si analizza l'adozione responsabile di nuovi framework, mettendo in guardia contro il rischio di "sprecare tempo e risorse" su tecnologie non sostenibili a lungo termine, fino ad arrivare all'importanza cruciale delle code review, dove "non si tratta di imporre un proprio stile, ma di trovare un compromesso che possa essere accettato da tutti."

Scoprirete come l'evoluzione da programmatore junior a senior si manifesti nella capacità di valutare criticamente ogni decisione tecnica, dall'impatto di una singola condizione nel codice ("la perdita di spensieratezza e un insieme di paure regresse") fino alla gestione del debito tecnico e alla rimozione di codice obsoleto ("mantenere codice non usato è costoso e col tempo rappresenta un inutile debito tecnico").

Attraverso esempi concreti e riflessioni profonde, questo testo vi accompagnerà nel processo di maturazione professionale, evidenziando come ogni decisione

tecnica debba essere valutata non solo per il suo impatto immediato, ma anche per le sue conseguenze future sul sistema e sulle persone che lo manterranno.

DRAFT DRAFT DRAFT

"Puoi cambiare questa condizione nel codice? Cosa ci vuole?"



Cosa differenzia un programmatore senior da un programmatore junior? La capacità di valutare le conseguenze di una modifica al codice diranno in molti, in realtà è la perdita di spensieratezza e un insieme di paure regresse.

La paura dei senior

Un tempo ero molto più veloce nel cambiare un algoritmo, nella modifica di un parametro di una funzione, nella creazione o distruzione di flussi e così via. L'obiettivo era, prima di tutto, fare quello che chiedeva il cliente e solo in seconda battuta fare in modo che il codice fosse mantenibile o che la scelta presa fosse la migliore all'interno di una serie di scelte più o meno ponderate.

Poi sono invecchiato e ho iniziato ad usare i test per capire se il mio codice andasse bene anche dopo una modifica, ho iniziato a scrivere documentazione per capire perché avevo ragionato in un certo modo e ho iniziato a fare code review per migliorare la comprensione del codice e la sua manutenzione.

Questo nuovo approccio "complicato" il mio lavoro, o per meglio dire: ha allungato i processi che, da una richiesta, portavano alla soluzione finale.

Una modifica che un tempo facevo in 10 minuti, ora occupa una giornata intera, perché devo valutare tutti gli impatti che quella modifica può avere, far girare tutti i test automatizzati, eseguire qualche test manuale dove non sono riuscito ad automatizzare, scrivere la documentazione, effettuare il merge con il codice principale, aspettare il build e i test di integrazione, verificare che tutto funzioni e finalmente dichiarare l'issue risolta, sperando di non aver dimenticato qualcosa.

Gli anni passano ed è inevitabile che ci sia l'evoluzione dal bambino spensierato che effettua una push senza test, alla cariatide che aggiusta anche gli spazi quando deve aggiungere codice al branch main (no, "master" non si utilizza più).

Quello che una volta era un "sì" felice ed entusiasta,
cresce in un "sì, ma faccio anche questo",
piano piano diventa un "hi",

fino ad arrivare ad un "no"
o peggio ancora un "dipende".

I senior non sono cattivi

Non è per cattiveria che succede tutto questo, è per la consapevolezza di quanto gira attorno a una singola riga di codice. La consapevolezza dell'errore o degli impatti di una modifica e per uno sterminato regresso che troppe volte ha minato la certezza che una singola condizione possa essere cambiata senza problemi.

Questo non vuol dire che le modifiche non vanno fatte, vuol solo dire che vanno sempre valutati gli impatti, in modo da non creare problemi più grandi di quelli che si vogliono risolvere e che spesso, più si invecchia e più gli impatti si moltiplicano nella testa del programmatore.

Pensiamo al ciclo di vita di un software e alla sua progettazione. All'inizio sviluppare un prodotto è relativamente facile: vediamo il nostro obiettivo, abbiamo un'idea su come raggiungerlo, facciamo in modo che clienti e stakeholder (ogni tanto mettere un ingleseismo a caso aiuta a far sembrare un ragionamento intelligente) siano soddisfatti, e se abbiamo fortuna riusciamo a creare un team coeso sul prodotto sia dal punto di vista tecnico che umano. Questo non è sempre scontato perché a volte basta un singolo elemento fuori dal coro in grado di destabilizzare un intero gruppo. Una volta raggiunto questo obiettivo, quando occorre fare una modifica tutti i membri sono allineati e consapevoli di quello che serve realizzare.

Quando si esce da questa fase di euforia collettiva e si esce dalla fase creativa, il team di sviluppo inizia a cambiare rispetto alla sua forma originale. Contestualmente il cliente diminuisce il budget, perché si entra in una fase di manutenzione e se non si presta la dovuta attenzione si inizia a perdere la padronanza del prodotto.

No, non sono certamente i test che possono governare il prodotto,
ma sono le teste che lo hanno ideato.

Il turnover di progetto è un processo inevitabile, a meno che non vi troviate in un igloo in mezzo al polo, privi di connessione internet, e i poveri programmati che hanno realizzato il progetto non siano in grado di inviare un curriculum se non attaccandolo ad una foca (anche se esiste sempre la possibilità che il programmatore si dedichi alla pesca, sua grande passione sopita dal codice).

Quando si cambia un componente del team, si perde una parte di conoscenza del prodotto e si inizia a perderne la padronanza. A cascata ogni intervento diventa sempre più costoso, lungo e rischioso.

La manutenzione di un progetto cambia le regole?

In un normale ciclo di vita di un prodotto succede che alcune parti siano soggette ad aggiornamenti frequenti, ma molte altre parti, per un discorso di maturazione, carenza di richieste o stabilità, non abbiano necessità di modifiche. Rendere rarefatte le modifiche diminuisce la conoscenza del codice da parte del programmatore stesso che le ha realizzate. Questo comporta che la persona che ha scritto 2 anni prima quelle righe di codice è normale che non le ricordi e abbia necessità di riguardarle per ricordare i processi che ne avevano portato la realizzazione.

Più è verticale la modifica, più è poi difficile ricordarne la ragione: non ricordi perché all'interno di una connessione ad una risorsa esterna hai messo un parametro e non un altro, perché usi una suite di cifratura al posto di un'altra o magari hai un branch aperto da mesi, in attesa di una verifica con un cliente su una funzionalità urgente, che è diventata all'improvviso di scarsa importanza, ma ora occorre fare il merge in una baseline con 400 push in più e ti chiedi se ha più senso rifare tutto da capo o applicare le modifiche a un rebase.

In mezzo a questo tipo di interventi ci sono i lavori di tutti i giorni, le modifiche alle funzionalità, le correzioni di bug, le richieste di nuove funzionalità, le correzioni di nuovi bug che hai introdotto correggendo un bug, il codice che ha introdotto un collega con scarsa conoscenza del prodotto, i cambi di comportamento introdotti dall'uso di un aggiornamento o un nuovo componente che obbliga un cambio di codice e introduce un'anomalia indiretta.

Tu che hai scritto quel codice non lo ricordi, capisci che a fronte di una serie di cambiamenti la modifica da fare è più grande di quanto prospettato, ma devi capire come farla, perché hai un ricordo dei razionali per i quali è stato fatto il codice precedente e capisci che non è una modifica banale. Figuriamoci chi deve sistemare lo stesso codice ed è la prima volta che mette le mani su quella parte di programma.

Come diceva Eraclito:

Nessun uomo può attraversare lo stesso fiume due volte,
perché né l'uomo né il fiume sono più gli stessi

Questa frase può essere facilmente riapplicata al codice: ogni modifica, anche la più piccola, cambia il codice e il contesto in cui si trova, che è diverso dal contesto in cui era stato realizzato e anche la persona è differente rispetto alla persona che aveva scritto quel codice.

Caso d'uso di una nota Banca italiana

Per un attimo immaginate di essere un programmatore senior di una "banca qualsiasi", dove vi è stato detto che si doveva procedere ad un aggiornamento di sistema e relativo firmware, che non avrebbe portato a nessun danno e che si sarebbe proceduto a fare l'aggiornamento in produzione.

Avendo visto questo tipo di interventi più volte nella vostra vita, la prima cosa che vi viene in mente di fare è quella di eseguire un test, ma non esiste un ambiente di test che copra esattamente la dimensione della produzione, sia per

dimensionamento che per casi d'uso. Sconsigliate l'aggiornamento massivo, ma il produttore della soluzione assicura che non ci saranno problemi, il reparto di sicurezza vi avvisa che occorre mettere l'aggiornamento entro una certa data perché altrimenti non si è conformi alle regole aziendali e che comunque in caso di problemi si può tornare indietro.

Vostro malgrado accettate, spinti anche dal fatto che "lavoriamo con la metodologia agile e dobbiamo essere pronti al cambiamento" e che "non possiamo essere sempre negativi".

Le interruzioni agli accessi ai servizi online (come app, app Invest, Internet Banking e Smart Business) si sono verificate in seguito all'installazione di un aggiornamento del sistema operativo e del relativo firmware che ha comportato una situazione di instabilità.
Ci teniamo ancora a scusarci e riteniamo fondamentale ringraziarti ancora per la comprensione dimostrata.

Un "banale" aggiornamento di sistema operativo e firmware ha comportato una situazione di instabilità per ben 5 giorni.

Chiaramente, in questo caso ipotetico, il problema non è stato l'aggiornamento, ma la mancanza di test, la mancanza di un ambiente di test che coprisse la produzione, la mancanza di un rollback immediato e la mancanza di un piano di disaster recovery o la sottovalutazione del rischio.

Qual è la percezione della modifica all'esterno del progetto?

Spesso chi è fuori dal progetto ha una percezione completamente diversa dello sviluppo software e spinge per realizzare le modifiche il prima possibile.

Mi serve per ieri

Questa frase è un classico, ma spesso non si capisce che una modifica, anche

banale, può comportare una serie di problemi che vanno ben oltre la modifica stessa.

Ho chiesto solo di cambiare una condizione, perché Mario mi crea tutti questi problemi?

Ora assegno il compito a Bruno che lo risolve in 2 minuti

Ci sono situazioni all'interno dei progetti che permettono di alterare il codice molto velocemente, ed altre situazioni in cui una modifica, anche banale, può richiedere molte riflessioni: anche se si tratta di poche righe di codice o addirittura una sola condizione aggiuntiva.

Ogni programmatore affronta la modifica in modo diverso, in base alla sua esperienza, alla conoscenza del prodotto e alla conoscenza del codice. Più abbiamo programmati esperti del prodotto, più è probabile che la modifica venga valutata attentamente, perché si è consapevoli di quanto una modifica possa comportare. Meno il programmatore è a conoscenza del prodotto, più è probabile che la modifica venga fatta velocemente.

Conclusioni

Ogni modifica, anche la più banale, può comportare dei problemi prestazionali, di regressione, di sicurezza, di manutenibilità, di comprensione del codice, di documentazione, di test, di deployment.

Allo stesso modo delle alterazioni esterne al progetto possono creare gli stessi problemi: ho aggiornato il sistema operativo, ho aggiornato un driver, ho aggiornato il firmware e ora non funziona più nulla.

La pressione che viene messa addosso ad un programmatore aumenta progressivamente col tempo perché parallelamente aumenta la consapevolezza di tutto quello che può andare storto e di quanto sia difficile risolvere un problema una volta che si è verificato.

Questo può portare a situazioni in cui un programmatore senior si rifiuta di fare una modifica, perché sa che i rischi sono troppo alti, o perché sa che la modifica richiederebbe troppo tempo e risorse per essere fatta correttamente.

La prossima volta che guardate un programmatore, che da molti anni lavora sullo stesso codice e non vuole modificarlo e vi avverte su ogni alterazione interna o esterna al progetto, non pensate a lui come a un incapace, ma piuttosto come a una persona consapevole ed utilizzate il suo consiglio per evitare di far crollare il vostro progetto e sopesare al meglio rischi e benefici di ogni modifica perché infondo, come diceva Isaac Asimov:

Nessuna decisione sensata può essere più compiuta senza tenere conto non solo del mondo come è ora, ma di come sarà

L'adozione di nuovi framework potrebbe far fallire il tuo progetto



Nel mondo dello sviluppo software, l'attrazione per le novità è sempre forte. È allettante mettere le mani su un nuovo prodotto, magari il primo a risolvere una serie di problemi che gli altri framework non riescono ad affrontare.

Questi nuovi progetti spesso vantano una comunità piccola ma energica, promettono di accelerare i tempi di sviluppo e presentano innovazioni interessanti rispetto ai concorrenti. Tutto ciò sembra molto promettente, ma è davvero la scelta giusta per un progetto a lungo termine?

Donald Knuth, uno dei padri dell'informatica moderna, commentava il proprio codice in questo modo:

Fate attenzione ai bug nel codice soprastante; ho solo dimostrato che è corretto, non l'ho testato.

Se fosse stata scritta da un programmatore junior, questa frase potrebbe sembrare una battuta. Ma detta da Knuth, oltre a essere umoristica, potrebbe anche segnalare che quella parte di codice non era testabile, ma solo sintatticamente corretta.

A volte è molto più semplice dimostrare un concetto, descriverne il funzionamento e documentarlo, rispetto alla sua reale applicazione, che potrebbe coinvolgere una serie di aspetti non considerati dalla dimostrazione.

Questo concetto si può parafrasare così: "Non è detto che un'idea brillante funzioni sempre nella realtà". Pensando a ciò, non posso non considerare il mondo delle blockchain: sulla carta strumenti eccellenti, ma nella pratica spesso criticati per prestazioni e utilizzo delle risorse.

Uscendo dal mondo software, pensiamo a prodotti come Segway che, nonostante fosse un'idea brillante, non ha avuto il successo sperato. O a Google Glass, che nonostante fosse un prodotto innovativo, non è mai decollato.

E i framework?

I framework e, più in generale, le tecnologie software, soffrono spesso di un hype iniziale che galvanizza i programmatori, distraendoli da una solida analisi della loro efficacia nel lungo termine. I concetti utilizzati in quel nuovo software sono applicabili a tutti i progetti? Sono facilmente integrabili con altre tecnologie? Ci sono solide aziende che investono in quella tecnologia? La comunità è attiva e supporta il prodotto?

I cosiddetti "early adopters" sono spesso attratti dalle novità, ma frequentemente non si rendono conto dei rischi che questa scelta comporta. Se va tutto bene, hanno fatto una scelta fortunata che ha portato al successo il proprio software. Ma se il prodotto scelto non riesce a mantenere le promesse, si ritrovano con un software che non funziona come dovrebbe e che richiede un lavoro di manutenzione e refactoring molto più pesante di quanto previsto.

Per chi sviluppa software da oltre 30 anni, il nome Visual Objects farà suonare qualche campanello: doveva essere il nuovo Clipper e portare milioni di programmatori DOS in ambiente Windows. Un'idea fantastica, ma che non ha funzionato: troppi problemi, troppi bug, troppi limiti. Ai tempi aveva alle spalle un'azienda come Computer Associates, che poteva permettersi di investire milioni di dollari in un prodotto, ma nonostante questo, non è mai decollato.

Ecco perché, quando si tratta di scegliere un framework o una tecnologia per un progetto, è importante considerare non solo le sue caratteristiche tecniche, ma anche la sua sostenibilità a lungo termine.



L'orizzonte temporale: una dimensione cruciale

Ogni volta che valuto un progetto, cerco di anticiparne il ciclo di vita, o almeno quello che dovrebbe essere, basandomi sull'esperienza e sulle informazioni a mia disposizione. La sfera di cristallo non esiste e il futuro non è prevedibile, ma ci sono segnali che fanno capire se un progetto sarà di breve, media o lunga durata.

Per comprendere quanto un progetto possa durare, è importante considerare vari aspetti:

- **Obiettivi a lungo termine:** Quali sono gli obiettivi del progetto? È un prodotto che risolve un singolo problema o qualcosa di strutturato per durare anni o solo pochi mesi?

- **Stabilità del mercato:** Il mercato in cui il progetto opera è stabile o in rapida evoluzione? L'effervesenza di un mercato può portare a cambiamenti rapidi che richiedono una maggiore flessibilità da parte del progetto.
- **Competizione:** Qual è il livello di competizione nel settore in cui opera il progetto? Lavorare in un'arena dove magari gli altri attori provengono da aziende più grandi e strutturate può richiedere una maggiore attenzione alla qualità e alla sostenibilità del progetto.
- **Tendenze del settore:** Quali sono le tendenze attuali nel settore in cui opera il progetto? Alcuni progetti nascono già vecchi, perché si basano su tecnologie obsolete o su modelli di business superati.
- **Risorse disponibili:** Quali risorse sono disponibili per lo sviluppo e la manutenzione del progetto? Si tratta di progetti "one man show" o ci sono team strutturati dietro?

Questi sono solo alcuni degli indicatori possibili, ma basarsi solo sul nome e su ciò che viene promesso è spesso una sottovalutazione del rischio.

Quando si utilizzano framework o tecnologie giovani, è importante considerare che potrebbero non essere supportati a lungo termine o potrebbero non essere in grado di adattarsi ai cambiamenti del mercato. Questo può portare a costi di manutenzione elevati, a problemi di compatibilità e a una maggiore complessità del codice.

D'altra parte, occorre anche valutare i progetti che si intendono realizzare. Potrebbero essere progetti nati e conclusi nel giro di un fine settimana: in questo caso l'esigenza primaria è la risoluzione del problema. Altri potrebbero dover reggere per pochi mesi per soddisfare un'esigenza specifica. Infine, esistono progetti destinati a durare anni, su cui un'azienda intende costruire la propria base operativa: in queste situazioni la scelta del framework e l'architettura di progetto diventa cruciale e non può essere basata solo sulla moda del momento.

L'orizzonte temporale è una dimensione che ogni architetto del software dovrebbe tenere in considerazione, perché influenza notevolmente le aspettative e le scelte che si possono fare.

Per un progetto "one-shot", si può tranquillamente adottare il framework CIRO, creato da un nomade della Tanzania, utilizzato solo dal suo gruppo di amici, con un unico rilascio fatto alcuni anni fa, se risolve in maniera sbalorditiva un problema cruciale per il progetto. Se invece devo realizzare qualcosa che deve durare oltre la scadenza di un dolce al cucchiaio, forse dovrei moderare la mia aggressività nella scelta dei componenti software da impiegare e basarmi su KPI differenti.

Il rischio dell'abbandono

Mi è capitato di usare prodotti sia "closed source" che "open source", il cui creatore è scomparso nel nulla, così come si è dissolta la comunità che ne faceva parte. Se ciò accade con prodotti "closed", è sicuramente molto preoccupante, ma non cadete nell'illusione che sia tutto rose e fiori se avviene con prodotti "open". Un conto è utilizzare un prodotto, un altro è svilupparlo e conoscerlo in ogni sua virgola.

La mente di chi progetta una soluzione è solitamente intrappolata tra le righe di codice del prodotto stesso ed è certamente più immersa nelle sue logiche interne rispetto a chi lo usa, che spesso sfrutta solo una parte delle sue potenzialità. Non pensate quindi che "open" significhi "c'è un problema: lo risolvo io", perché non è sempre così.

Linus Torvalds, creatore di Linux, ha un'opinione forte su questo argomento:

Il software è come il sesso: è migliore quando è libero e gratuito.

Anche se Torvalds utilizza questa citazione per promuovere il software open

source, è importante ricordare che "gratuito" non significa necessariamente "senza costi" e che la libertà di disporre dei sorgenti di un progetto non necessariamente significa poterlo mantenere e farlo evolvere.

L'importanza dell'analisi periodica

Questo è uno dei motivi per cui dovremmo sempre eseguire un'analisi della vita dei prodotti che adottiamo. Un'analisi da ripetere periodicamente, che tenga conto del contesto attuale, della maturità del progetto e delle sue prospettive future.

Se sui progetti brevi si può ignorare questo aspetto, su quelli che superano gli anni di sviluppo, il problema diventa incombente: o si aggiorna il prodotto e le sue dipendenze, o ci si trova in un vicolo cieco a causa delle innumerevoli intrecciature del proprio codice.

Un'analisi periodica ci permette di valutare se il framework o la tecnologia che stiamo utilizzando è ancora adatto alle nostre esigenze, se è supportato attivamente dalla comunità e se è in grado di adattarsi ai cambiamenti del mercato.

Il rischio dei servizi non standard

Questo si applica anche a tutti i problemi che possono scaturire dall'adozione di un servizio senza standard, che potrebbe essere ritirato dal mercato o raddoppiare i prezzi da una stagione all'altra.

Jeff Bezos ha una visione interessante su come le aziende dovrebbero approcciarsi alla tecnologia:

Bisogna essere testardi nella visione e flessibili nei dettagli

Questa filosofia può essere applicata anche alla scelta dei framework, dei servizi e

dello sviluppo software: mentre i dettagli tecnici possono cambiare, i principi di base come la scalabilità, la manutenibilità e l'interoperabilità dovrebbero rimanere costanti.

Per questo motivo, realizzare un software strettamente accoppiato con un framework potrebbe portare a problemi di manutenzione e scalabilità in futuro. Se il framework non è più supportato o non è in grado di adattarsi ai cambiamenti del mercato, potremmo trovarci in una situazione in cui è necessario riscrivere gran parte del codice per adattarlo a una nuova tecnologia.

Ho visto prodotti fare scelte tecniche che, sulla carta, sembravano le migliori, e col tempo accorgersi che l'accoppiamento software che era stato realizzato era talmente forte che cambiare una parte significava dover cambiare tutto il resto. Anche questo problema si può evitare con una corretta analisi e una scelta ponderata dei componenti utilizzati, disaccoppiando utilizzo da implementazione e accettando di dover cambiare una parte del software se necessario.

Conclusioni

Quando si tratta di soluzioni o framework giovani, il rischio di sprecare tempo e risorse è considerevole. Se amate l'incertezza e la durata del progetto non è un fattore che volete considerare, siete liberi di fare qualsiasi scelta e di costruire un prodotto ricco di domande irrisolte.

In caso di errori, tenete presente che potrebbe essere necessario ricominciare da zero, a meno che il vostro software non sia sufficientemente modulare da permettere la sostituzione di un componente senza dover riscrivere tutto il resto.

La scelta di un framework o di una tecnologia dovrebbe essere guidata non solo dall'entusiasmo per l'innovazione, ma anche da una valutazione ponderata dei rischi e dei benefici a lungo termine. La sostenibilità, la manutenibilità e la scalabilità dovrebbero essere sempre in cima alle nostre priorità quando prendiamo decisioni architetturali.

Il Mito del Full Stack Developer: una realtà scomoda



Il termine "Full Stack Developer" ha subito una trasformazione radicale che merita un'accurata riflessione.

Iniziamo con la definizione di cosa significhi essere un "Full Stack Developer", attingendo da ChatGPT:

I "full stack developer" sono sviluppatori software che hanno competenze sia nel front-end che nel back-end dello sviluppo web o di applicazioni software. In altre parole, sono in grado di lavorare su tutte le parti di un'applicazione o di un sito web, dal lato client che viene eseguito nel browser dell'utente (front-end) al lato server (back-end) che gestisce la logica di business e l'accesso ai dati.

Il mio approccio col termine Full Stack Developer

In un determinato periodo della mia carriera, questa definizione rispecchiava perfettamente il mio stato d'animo, era ciò che sentivo di essere e ciò che desideravo che gli altri percepissero quando mi osservavano. Orgoglioso di questo status, inserivo tale qualifica nel mio curriculum vitae, quasi come quando si ottiene un voto eccellente a scuola e si desidera condividerlo con la propria famiglia.

Col trascorrere del tempo ho scoperto che il fascino di questa definizione ha conquistato il cuore di molte persone, principalmente a causa di un pensiero comune: il Full Stack è un programmatore completo e, per le aziende, assumere un Full Stack Developer rappresenta un valore aggiunto.

Se assegno a questo tipo di programmatore un progetto, lui è in grado di gestire in modo autonomo tutti gli aspetti dello sviluppo software: è come avere un intero team di sviluppo racchiuso in un'unica persona!

Col tempo mi sono accorto che questo approccio poteva avere un senso quando i progetti erano semplici e le tecnologie poco numerose.

Oggi il mondo è cambiato, è molto più complesso coprire ogni sfaccettatura di un progetto. Ora essere un "Full Stack Developer" sembra essere sinonimo di lavorare in modo mediocre su tutti gli aspetti di un progetto: non si è un esperto di backend, né di frontend, né di devops; si conoscono le nozioni di UX, UI, SEO, marketing, si conosce un po' di tutto, ma nulla in modo approfondito, o magari solo una parte.

Proviamo ad traslare il concetto di "Full Stack" sul vostro meccanico: il vostro meccanico, quando c'è un problema alla carrozzeria, non la ripara a martellate, ma coinvolge un carrozziere; quando ci sono problemi di carburazione chiama un esperto di carburatori; quando c'è un problema elettrico, coinvolge un elettrauto.

Allo stesso modo, se il vostro medico sospetta un particolare problema, vi manda da un esperto specializzato in quell'area, che vi sottoporrà a una visita approfondita per formulare una diagnosi e una cura mirata.

Un programmatore "Full Stack" no: è stato scelto per non dover chiamare nessuno, per risolvere tutto da solo.

Questo approccio non è sostenibile, o meglio non è affrontabile pensando che il Full Stack sia un punto di arrivo e non un punto di partenza.

Molte professioni ci fanno comprendere che l'esperto di un particolare ambito è preferibile a un 'tuttologo', se vogliamo svolgere un lavoro a regola d'arte.

I programmatori esperti sanno bene quanto sia importante specializzarsi e riconoscono le ore perse nella ricerca di soluzioni che un esperto avrebbe risolto rapidamente.

Nella cultura di massa, e spesso quando si parla con i propri committenti, vi è la presunzione che un programmatore debba essere in grado di fare tutto alla perfezione.

Rispondere con frasi come 'non è il mio campo, ci vorrebbe un esperto di UI' viene

visto come un segno di debolezza, pur essendo una risposta legittima, professionale e onesta.

- "Sono un Full Stack Developer"
- "Ah, quindi sei un esperto di tutto?"
- "No, non proprio"

Il mercato del lavoro e il Full Stack Developer

Il mercato del lavoro ha iniziato a cavalcare l'onda della richiesta di "Full Stack Developer", rispondendo a questa esigenza in modi talvolta bizzarri.

Stanno emergendo figure come i "Full Stack Developer Junior", che si dichiarano "full stack" ma aggiungono il termine "junior" per sottolineare la loro inesperienza. Spesso, questo titolo segue corsi intensivi del tipo "diventa full stack in 6 mesi".

Non ce ne rendiamo conto, ma ci troviamo di fronte a un problema serio. Da un lato, stiamo mettendo nella testa dei giovani programmatore l'idea di poter coprire autonomamente e facilmente l'intero stack tecnologico, per poi farli scontrare con la cruda realtà: al primo progetto complesso che incontrano, si rendono conto di non essere in grado di gestirlo da soli.

Non è però tutta colpa di questi nuovi programmatore. Esiste anche un problema legato ad alcune aziende che non hanno la capacità di valutare accuratamente il reale livello di competenza delle persone ed assegnano ruoli di responsabilità basandosi sulle poche informazioni in loro possesso.

Rispondere in modo brillante a un colloquio tecnico è una cosa, ma come sostiene saggiamente Linus Torvalds:

Talk is cheap. Show me the code

La qualità del codice e la valutazione del programmatore

Il codice, l'approccio alla risoluzione dei problemi e il modo in cui si affrontano le difficoltà sono gli aspetti che contraddistinguono un programmatore dall'altro, anche se entrambi si definiscono "Full Stack Developer".

Purtroppo, valutare quanto una persona sia effettivamente in grado di produrre non è affatto semplice. Ho visto programmati produrre grandi quantità di codice, ma di scarsa qualità, e programmati che ne producevano poco, ma di altissima qualità.

Per voler citare Antoine de Saint-Exupéry:

La perfezione si raggiunge non quando non c'è più nulla da aggiungere, ma quando non c'è più nulla da togliere

Riuscire a valutare il vero valore di un programmatore è un'abilità molto difficile da acquisire, eppure è una competenza fondamentale per un buon manager. Non è sufficiente valutare il numero di righe di codice prodotte, né il numero di progetti completati o di tecnologie conosciute, soprattutto in un'era in cui le tecnologie cambiano rapidamente e ci aiutano a scrivere sempre più codice in meno tempo.

La quantità di codice non è un indicatore di qualità, anche nel caso in cui funzioni perfettamente e sia coperto da numerosi test funzionanti. Il codice non deve essere valutato al chilo, bensì per la sua qualità, manutenibilità, scalabilità e leggibilità.

Chi può valutare la qualità del codice? Non è facile rispondere a questa domanda.

La risposta più semplice potrebbe essere: un altro Full Stack Developer, ma questa non è la risposta giusta.

Non lo è perché un Full Stack Developer non è un esperto in tutto, ma dovrebbe essere esperto soprattutto nel suo ambito applicativo specifico. Quindi la valutazione del valore di una persona da parte di un Full Stack Developer

potrebbe essere influenzata da una serie di bias dovuti alle sue esperienze personali, conoscenze pregresse e competenze acquisite.

Allora, può essere un HR a valutare la qualità del codice? O un manager? Un collega? Un cliente? La risposta giusta è: tutti loro possono contribuire.

Anche in questo caso, però, la valutazione sarà data dalla somma delle esperienze delle varie persone coinvolte, che potrebbero avere opinioni diverse e decretare che lo stesso Full Stack Developer sia un genio in uno stack tecnologico come LAMP, ma un incapace in un altro come MEAN.

Se lavorassimo invece sulla "problem solving attitude"?

Abbiamo quindi metabolizzato che "Full Stack Developer" è un termine non qualificante, interpretabile in modo diverso a seconda del contesto e, soprattutto, molto oneroso dal punto di vista degli aggiornamenti tecnologici necessari.

Proviamo quindi a lavorare su un concetto diverso: la "problem solving attitude".

Un programmatore che possiede una solida "problem solving attitude" è in grado di affrontare qualsiasi problema, anche se non ne ha mai affrontato uno simile in passato. È capace di analizzare il problema, scomporlo in parti più piccole, trovare una soluzione, implementarla e testarla con successo.

Questo tipo di programmatore non si lascia scoraggiare dalle difficoltà, ma le affronta con determinazione, adattandosi e imparando lungo il percorso. Questa mentalità flessibile e analitica è fondamentale per affrontare le sfide sempre nuove che si presentano nello sviluppo software.

Un programmatore con una solida "problem solving attitude" non ha timore di chiedere aiuto, di ammettere "non so" o "non ci riesco". Anzi, è in grado di lavorare efficacemente in team, condividendo le proprie conoscenze, imparando da chi ne sa di più e insegnando a chi ne sa di meno.

In questo contesto, un grande bagaglio di esperienze in contesti e tecnologie diverse, in progetti internazionali con tecnologie eterogenee, è sicuramente un valore aggiunto, molto più che essere un Full Stack Developer. Quest'ultimo, infatti, non si specializza nella risoluzione di problemi, ma nella conoscenza approfondita del proprio stack tecnologico specifico, risultando fuori contesto come un giocatore di bowling in una partita di briscola.

Riuscire a risolvere problemi anche in contesti non familiari è un valore aggiunto prezioso all'interno di un team. Ci sarà sempre tempo in seguito per perfezionare la soluzione mettendola nelle mani di un esperto di prodotto, ma la capacità di risolvere problemi in modo autonomo è una competenza inestimabile.

Un programmatore con una mentalità flessibile, analitica e collaborativa, unita a un'ampia gamma di esperienze diverse, sarà in grado di affrontare con successo le sfide più complesse, adattandosi e crescendo costantemente.

Conclusioni

Il mio consiglio è di evitare di dichiararsi "Full Stack Developer".

Questo termine è ormai troppo inflazionato e, scritto sul CV di un programmatore junior, non aggiunge alcun valore, se non l'ilarità dei programmatori più esperti che lo leggono.

È molto meglio identificare in modo preciso le tecnologie in cui si è esperto e quelle in cui si ha avuto meno esperienza. Per tutti i programmatori è così: puoi essere un mago di Angular, aver scritto delle API REST in Node, ma se un esperto di Backend analizza il tuo codice, probabilmente troverà delle lacune.

Esponendo con chiarezza quali sono le proprie competenze ed esperienze, sarà poi il tuo interlocutore a valutare il tuo effettivo valore, un aspetto che vale molto più di una generica etichetta di "Full Stack Developer".

Occorre concentrarsi sul costruire una solida "problem solving attitude", sull'acquisire esperienza in contesti tecnologici diversi, uscendo dalla propria comfort zone, promuovendo un approccio collaborativo al lavoro.

Queste qualità saranno molto più apprezzate di un'autocertificazione come "Full Stack" che rischia di risultare fuorviante o eccessivamente generalista.

"Vediamo chi c'è l'ha più grosso: facciamo code review"



La revisione del codice è uno dei momenti più delicati e importanti di un progetto software. Qualsiasi progetto software che abbiamo sviluppato o svilupperemo nel tempo è stato o sarà sottoposto a una revisione del codice.

Non è immaginabile un codice scritto da chiunque non subisca mutazioni nel tempo. Anche se può valere per singole parti di un progetto, se guardiamo l'interesse di un progetto software, la code review diventa un passaggio significativo fra rimanere piccoli e giocare con le bambole, o uscire allo scoperto e diventare uomini.

Questo passaggio è quindi fondamentale per garantire la qualità del codice, per evitare che errori possano compromettere il funzionamento del software e per allargare la conoscenza del codice a tutto il team. Sì, perché far evolvere il team è un aspetto più importante che sublimare il proprio ego nel guardare una decina di righe che solo chi le ha scritte può capire.

La code review da fastidio

Esiste però un problema di fondo: ai programmatori la code review infastidisce, non piace, viene ritenuta superflua o sbagliata se fatta da altri e deve essere fatta e decisa solo da loro.

Per alcuni poi è come perdere un figlio:

Quel codice era scritto in quel modo per una ragione ben precisa: Mario non voleva che nessuno ci mettesse le mani, ma ora che si è licenziato non si capisce più nulla.

Un buon momento per fare code review è quando si corregge un bug. Occorre però scardinare l'approccio che viene spesso sposato dai programmatori, che tendono ad essere chirurgici e a risolvere la singola segnalazione, senza sfruttare il pensiero olistico.

Mi hanno segnalato che cliccando su un pulsante c'è un errore, faccio in modo che non ci sia.

In realtà, il problema potrebbe essere molto più profondo e richiedere una riflessione più ampia. Ci dovremmo porre il problema di come mai quel pulsante è stato cliccato, se è stato cliccato da un utente, se è stato cliccato da un utente che non avrebbe dovuto cliccarlo, se è stato cliccato da un utente che non avrebbe dovuto cliccarlo in quel momento, se è stato cliccato da un utente che non avrebbe dovuto cliccarlo in quel momento e in quel contesto.

Volendo allargare il pensiero, potremmo anche chiederci se il problema non è esteso ad altri pulsanti, se e come viene segnalato, se ha senso avere quel pulsante o quei pulsanti che fanno quell'operazione, se l'operazione che fa quel pulsante è necessaria, se l'operazione che fa quel pulsante è necessaria in quel contesto.

Allargare il contesto però vuol dire spendere tempo, impiegare risorse, mettere in discussione il proprio lavoro e quello degli altri, vuol dire mettere in discussione il proprio ego e quello degli altri, ma così facendo si migliora il codice, si migliora il sistema, si migliora il team.

Occorre quindi non guardare la singola segnalazione che ci viene fatta, ma cercare di capire il contesto in cui è stata fatta e se ci sono altre parti di codice che potrebbero essere migliorate.

Il pensiero olistico, anche nel codice, è importante: se non capiamo che una singola modifica ha un impatto su tutto il sistema, non possiamo capire il sistema, se non capiamo il sistema, non possiamo migliorarlo.

Come diceva Platone:

Il refactoring è l'arte di togliere il male e aggiungere il bene al codice.

Forse l'autore non era Platone, ma il concetto è chiaro: il codice va migliorato, va reso più leggibile, più mantenibile, più testabile.

Il codice non usato

Se avete lavorato sullo stesso progetto per più di un lustro, la frase:

Quel codice lo abbiamo scritto per "Cliente importante", che non è più nostro cliente, ma non lo togliamo perché "potrebbe" servire

è quasi un mantra che sono sicuro avete sentito almeno una volta.

Stratificare pensieri di questo genere nel corso del tempo porta ad avere un codice che non si capisce, che non si sa cosa faccia, che non si sa perché lo fa e che non si sa come lo fa: le specifiche erano intrecciate con i bisogni del cliente, con le esigenze del team, con le scelte tecniche del momento.

Nel frattempo, il codice è cambiato, il team è cambiato, il cliente è cambiato, le esigenze sono cambiate, le scelte tecniche sono cambiate: è giusto che il codice cambi o venga rimosso dal progetto. Mantenere codice non usato è costoso e col tempo rappresenta un inutile debito tecnico.

Ci sono aziende che hanno capito quanto sia costoso e inutile mantenere progetti che non vengono più usati: il codice è un costo, non un valore, se non viene usato.

Google, ad esempio, ha una ricca storia di progetti che sono stati abbandonati. Ne potete trovare traccia su Wikipedia:

https://en.wikipedia.org/wiki/Category:Discontinued_Google_services

Io uso i TAB, chi usa gli spazi non è un programmatore bravo

In una puntata di "Silicon Valley", il protagonista Richard scopre che la sua fidanzata Winnie utilizza gli spazi al posto dei TAB per indentare il codice e la lascia: è un'esagerazione cinematografica, ma conosco persone che non accetterebbero mai di lavorare con qualcuno che usa gli spazi al posto dei TAB e viceversa.

Scardinare questa abitudine, e lo dice uno che usa gli spazi da quando ha appoggiato l'indice sulla tastiera, è difficile, ma necessario: il codice deve essere uniforme, deve essere coerente, deve essere leggibile. Avere stili diversi all'interno del team è controproducente, rallenta il lavoro, aumenta la possibilità di errori, rende difficile la manutenzione.

All'interno di una code review è necessario discutere anche di queste cose: non si tratta di imporre un proprio stile, ma di trovare un compromesso che possa essere accettato da tutti e possibilmente non sia un peso per nessuno.

Una volta raggiunto un accordo, è importante rispettarlo: se si decide di usare gli spazi al posto dei TAB, non si può tornare indietro.

Si tratta però di qualcosa difficile da accettare e da mettere in campo, sia perché alcuni programmatore considerano il codice come un figlio, che non si può cambiare, sia per il fatto che, anche imponendo uno stile uniforme, prima o poi qualcuno non capirà la specifica, altri configureranno l'IDE in modo sbagliato o perderanno le impostazioni al primo aggiornamento.

In queste situazioni, dove non è il linguaggio stesso che ci aiuta a mantenere uno stile uniforme (sì, ci sono linguaggi che lo fanno per noi in modo assolutamente dittoriale), è importante utilizzare degli strumenti che ci permettano di automatizzare il più possibile la formattazione del codice.

All'interno di uno dei progetti più grossi che seguo, è stato deciso di adottare OpenRewrite, uno strumento che riformatta il codice in modo automatico, uniforme e coerente, riducendo le differenze fra diversi stili di programmazione:

uniformare, oltre a rendere il codice più coerente, velocizza l'inserimento di nuovi membri nel team e il passaggio di un componente da un team all'altro.

Resistenza al cambiamento

Anche se la maggior parte dei programmatori è d'accordo sulla necessità di una code review, spesso si rifiutano di accettare i cambiamenti proposti, un po' per pigrizia, un po' per orgoglio, un po' per abitudine.

Se lavori da anni nello stesso modo, e funziona, perché cambiare?

Il mio codice nasce perfetto

Purtroppo no: non è così. All'interno di un team, l'aspetto che deve prevalere è la lettura del codice da parte di tutti, deve prevalere la facilità di inserimento di una nuova risorsa, deve prevalere la semplicità di modifica anche dopo mesi che non si mette mano ad alcune righe.

Non importa quindi se l'IDE che usi formatta il codice in un certo modo, non importa se ti è sempre piaciuto dichiarare le variabili con "_", aderire ad uno standard diffuso, spontaneamente o tramite delle automazioni, è un vantaggio, anche se in prima battuta può sembrare un peso.

Clean Code

Uno degli obiettivi che dovrebbe avere una code review è quello di scrivere codice pulito, leggibile e mantenibile.

A volte mi sono trovato di fronte alla scelta se mantenere del codice che funzionava, o smontarlo perché, pur superando in modo eccellente qualsiasi test funzionale, era scritto in modo oscuro, difficile da leggere e da mantenere, e ogni volta che mettevo una persona davanti a quel codice, la sua espressione era sempre la stessa:

Non capisco cosa fa

A volte crediamo che usare l'ultima sintassi proposta da un linguaggio sia la soluzione a tutti i nostri problemi, ma non sempre è così. A volte i nuovi costrutti si basano su concetti difficili da comprendere e da spiegare, a volte sono solo zuccherini per farci sentire più bravi: "sugar code" come lo chiamano alcuni.

Non sempre il codice più compatto e conciso è il migliore: a volte è meglio scrivere 10 righe di codice che fanno quello che devono in modo chiaro e leggibile, piuttosto che una sola riga che fa la stessa cosa, ma che nessuno capisce, per guadagnare nulla a livello prestazionale e funzionale.

Automatizzare il possibile

Stranamente, i programmatore accettano di buon grado qualcosa che viene proposto da un programma, rispetto a quanto viene proposto da qualcuno all'interno del team.

Se anche nel vostro team c'è lo stesso clima, potete proporre l'introduzione, all'interno della pipeline di progetto (sperando ne abbiate una), di processi di normalizzazione di codice e analisi statica.

I primi serviranno a diminuire la differenza di stile fra un programmatore e l'altro, gli altri serviranno a suggerire modifiche, a farci capire dove sono nascosti bug, dove mancano dei test, dove la complessità cognitiva è troppo alta o dove possiamo ridurre il codice, senza perdere di chiarezza o efficacia.

Ma a me non interessa

Ogni tanto passo il mio tempo libero a studiare progetti di altri programmatori, faccio girare qualche analizzatore e provo a correggere un po' di codice: è un buon allenamento e mi permette di vedere come altri affrontano i problemi o

sottovalutano le conseguenze del loro codice.

Durante queste scorribande, mi sono imbattuto in approcci completamente differenti da parte di singoli programmatore o team di programmatore ai quali ponevo le mie Pull Request.

Fra tutti i progetti che ho analizzato, vorrei parlare di due che mi hanno particolarmente colpito per come il team ha gestito il mio contributo.

La prima Pull Request riguarda il JDK di OpenJDK. Mi sono accorto che, per distrazione, 82 sorgenti contenevano dei doppi ":" alla fine di una riga.

Capite che non si tratta di qualcosa di importante, possiamo anche dire che si tratta di qualcosa di trascurabile, ma ho deciso di fare una Pull Request per correggere il problema e capire come si sarebbe comportato il team di sviluppo.

Potete trovare la PR a questo indirizzo:

<https://github.com/openjdk/jdk/commit/ccad39237ab860c5c5579537f740177e3f1adcc9>

L'approccio, a mio avviso, è stato interessante: prima di tutto sono stati coinvolti nell'analisi della modifica 8 persone, essendo le modifiche orizzontali a tanti package di Java.

Dopo una discussione orizzontale di vari maintainer e aver constatato che il problema era reale, è stata aperta una issue:

<https://bugs.openjdk.org/browse/JDK-8282657>

Si è anche discusso di una possibile modifica ai tool di build, che già rimuovevano gli spazi a fine riga, introducendo la rimozione dei doppi caratteri ":".

Questo approccio denota un team che analizza ogni singola modifica, che discute

e che cerca di capire se la modifica proposta è effettivamente utile o se è solo un capriccio di qualcuno.

In questo caso, hanno capito che si trattava di una modifica banale, ma orientata alla pulizia del codice, e per questo è stata accettata.

Un secondo caso invece coinvolge un tool di sicurezza realizzato da un singolo maintainer, messo anch'esso su GitHub.

In quel caso, avevo proposto una PR molto più seria. Era presente del codice ridondante, venivano utilizzate classi sincronizzate in processi nei quali non serviva una sincronizzazione, alcune risorse erano aperte senza una chiusura esplicita e così via.

Si trattava quindi di una modifica orientata a migliorare la qualità del codice e a ridurre la possibilità di errori, non qualcosa di banale come la rimozione di un carattere.

Oltre a questo, il passaggio a oggetti non sincronizzati portava a un aumento prestazionale nell'intorno del 20%: tutto questo senza snaturare il codice, ma solo utilizzando i giusti costrutti e, a parità di interfacce, le giuste classi.

In questo caso, ero convinto che non ci fossero problemi a inserire questo codice nel progetto, ma la risposta è stata negativa.

Il maintainer ha risposto che non intendeva integrare nel suo codice i suggerimenti che derivavano da un analizzatore sintattico e che potevo usarli sul mio fork e lui non li avrebbe mai integrati.

Se non ti piace il mio codice, fai il tuo fork e modificalo come vuoi

Questa è la classica risposta di chi non vuole cambiare, di chi non vuole accettare che il suo codice possa essere migliorato, di chi non vuole accettare che il suo codice possa essere cambiato da altri, nonostante le modifiche possano

migliorare la leggibilità e portino a tangibili miglioramenti.

Conclusione

Anche quando tutto funziona, revisionare il codice è importante: permette di migliorare il progetto, di migliorare il team e tagliare quelli che sono i debiti tecnici.

Diminuire il debito tecnico, avere il coraggio di smontare anche codice funzionante, buttare codice non più usato, dare ascolto a revisori statici, uniformare lo stile di programmazione, automatizzare il più possibile la formattazione del codice, sono tutti passaggi che possono migliorare il codice e il team.

Non abbiate paura di mettere in discussione quello che è stato scritto in passato e allargate la visione quando dovete mettere le mani al codice: non guardate solo la singola modifica, ma cercate di capire il contesto in cui è stata fatta e se ci sono altre parti di codice che potrebbero essere migliorate.

Un prodotto migliore passa attraverso tanti piccoli passi e non è detto che, pur avendo centinaia di test funzionanti, il codice sia perfetto: la code review è un passaggio fondamentale per garantire la qualità del codice e per evitare che errori possano compromettere il funzionamento del software.

Il "celodurismo" dei programmatori



Nel mondo della programmazione, esiste un fenomeno curioso che potremmo definire "celodurismo". Questo termine descrive un atteggiamento di ostinata resistenza all'evoluzione degli strumenti e delle pratiche di sviluppo software.

Un tempo, quando le risorse erano scarse e preziose, i programmati dovevano essere incredibilmente ingegnosi per spremere ogni goccia di potenza dai loro sistemi. Questa necessità ha forgiato abitudini e mentalità che, in alcuni casi, sono sopravvissute ben oltre la loro utilità pratica, trasformandosi in una sorta di folklore professionale.

640K ought to be enough for anybody

Questa celebre frase attribuita a Bill Gates, anche se lui ne ha negato la paternità, rappresenta bene l'atteggiamento di molti "celoduristi" nei confronti dell'innovazione tecnologica. Per questi programmati, l'idea di utilizzare strumenti moderni come IDE avanzati, debugger grafici o assistenti AI sembra quasi un'eresia, una violazione dei principi fondamentali della programmazione "vera", un sintomo di debolezza e incapacità.

Le radici storiche

Negli albori dell'informatica, quando i computer venivano alimentati a colpi di schede perforate, programmare significava lavorare con sistemi dotati di memoria e processori limitati. I programmati dell'epoca dovevano essere veri e propri artisti dell'ottimizzazione, capaci di scrivere codice che fosse al contempo funzionale ed estremamente efficiente.

Questa era ha prodotto capolavori di ingegnosità, e pratiche che ancora sopravvivono in ambienti dotati di scarse risorse: avete mai provato a programmare per microcontrollori o sistemi embedded? Qui, ogni byte di memoria e ogni ciclo di clock contano, e l'arte dell'ottimizzazione vive ancora di ottima salute, anche se più di una crepa inizia a vedersi all'orizzonte.

Col passare del tempo e l'evoluzione dell'hardware, molte di queste limitazioni sono venute meno, ma l'etica del "fare di più con meno" è rimasta profondamente radicata nella cultura della programmazione, talvolta trasformandosi in un atteggiamento di resistenza verso strumenti e pratiche che potrebbero semplificare e velocizzare il lavoro di sviluppo.

"Il codice più efficiente è quello che non devi scrivere"

Il manifesto del celodurismo

Per capire che tipo di programmatore siete, ecco un piccolo test per valutare il vostro grado di "purezza".

Provate a capire quanto siete a favore o a sfavore di queste affermazioni:

1. L'IDE non serve a nulla

Un programmatore deve essere spartano: meno strumenti usa, più è sinonimo di competenza

"Per scrivere del codice mi basta Notepad (o Vi) e la CLI"

Questa affermazione è spesso pronunciata con un misto di orgoglio e nostalgia. Chi la sostiene sembra voler comunicare: "Sono un vero programmatore, non ho bisogno di aiuti". Questa posizione ignora deliberatamente i vantaggi offerti dagli IDE moderni.

Personalmente ho visto programmatori usare comandi come:

copy con: pippo.prg

con lo stesso orgoglio di chi a scuola prende un ottimo voto.

Non sottovalutiamo però i vantaggi degli IDE moderni e come come prima cosa

leviamoci dalla testa che gli IDE siano dei semplici editor di testo con qualche abbellimento. Sono potenti strumenti che integrano numerose funzionalità per migliorare la produttività e la qualità del codice:

- Debugging: Punti di interruzione, ispezione e alterazione delle variabili in tempo reale, esecuzione passo-passo del codice, call stack e molto altro.
- Refactoring: Rinominare variabili o funzioni in tutto il progetto con un solo click, estraendo metodi o riorganizzando il codice in modo sicuro.
- Analisi statica: Identificazione di potenziali bug, violazioni di stile o pattern problematici prima dell'esecuzione.
- Integrazione con sistemi di versionamento: Gestione di branch, commit e merge direttamente dall'interfaccia dell'IDE.
- Supporto per framework e librerie: Autocompletamento, non delle sole librerie base, ma anche suggerimenti specifici per i framework utilizzati nel progetto.

Questi sono solo alcuni dei motivi per i quali, quando entro in un editor inventato negli anni '70, mi sento come se mi avessero tolto un braccio.

Per onestà intellettuale non possiamo negare che, in alcune situazioni, un IDE potrebbe essere eccessivo, quando ad esempio occorre fare una modifica rapida o quando si lavora su sistemi remoti con risorse limitate. In questi casi editor minimi o qualsiasi cosa che apre una console testuale, potrebbero essere la scelta migliore.

Per il lavoro di tutti i giorni e per progetti di una certa complessità, rifiutare categoricamente l'uso di un IDE moderno significa privarsi di strumenti che possono significativamente migliorare la qualità del codice e la produttività del programmatore e dell'intero team (si, perché il codice non è solo tuo, ma di tutti quelli che ci lavorano).

2. La CLI è la soluzione a tutti i problemi

Non nascondiamoci dietro a un dito, la CLI ha un fascino innegabile. Anni di film con hacker piegati su tastiere meccaniche e schermi neri con scritte verdi hanno formato generazioni di programmatori. Cosa c'è di più potente che controllare un computer digitando comandi testuali?

Questo approccio, anche se ha il fascino della prima ragazza incontrata al liceo, ha una serie di limiti che spesso vengono trascurati:

- Curva di apprendimento: Memorizzare pochi comandi è facile, memorizzarne decine o centinaia di comandi, con tutte le loro opzioni può essere scoraggiante per i neofiti e per chi non usa la CLI quotidianamente.
- Visualizzazione dei dati: Alcune informazioni sono semplicemente più facili da comprendere quando presentate graficamente.
- Operazioni complesse: Certe attività, come la gestione di branch in un repository Git, possono diventare molto più intuitive con una rappresentazione visuale.

Git è un ottimo esempio di come CLI e GUI possano coesistere e completarsi a vicenda. Mentre la CLI di Git offre un controllo granulare e la possibilità di automatizzare operazioni attraverso script, strumenti GUI come le integrazioni in VSCode possono rendere più accessibili operazioni complesse come:

- Visualizzare la storia dei commit con un grafico delle branch
- Effettuare rebase interattivi
- Risolvere conflitti di merge con strumenti di diff visuale

Come sempre la verità sta nel mezzo e l'approccio più saggio è quello di padroneggiare entrambi gli strumenti. Usare la CLI per operazioni rapide e per creare script automatizzati, ma non si deve esitare a passare a una GUI quando questa può offrire una visione più chiara o un flusso di lavoro più efficiente.

3. Non usiamo Windows perché i programmatori usano Linux (o MacOS)

Ormai non conto più le notti che ho passato a leggere e commentare le "guerre di religione" dei sistemi operativi.

"Winzoz" on funziona

Sarà bello Linux che ha 200 versioni e tutte incompatibili

Lasciate stare, con MacOS funziona tutto

Guarda che Apple ti fa pagare il doppio di quello che vale quel computer

Si potrebbe andare avanti all'infinito, creando flame pieni di odio e carichi di ignoranza, ma la realtà è che ogni sistema operativo ha i suoi pregi e difetti, ed è sbagliato sostenere che uno sia superiore agli altri in modo assoluto.

Un programmatore dovrebbe essere superiore a queste chiacchiere da bar. Questo non limita la sua libertà di sentirsi a suo agio su un sistema piuttosto che su un altro, ma è assolutamente controproducente mettersi i paraocchi e ignorare l'esistenza di altri sistemi operativi oltre a quello preferito.

Ragioniamo poi sul fatto di lavorare giornalmente su piattaforme differenti e dei vantaggi che questo approccio può portare:

- Cross-platform: Sviluppare e testare su più piattaforme assicura che il software funzioni correttamente per un'ampia base di utenti. "Write once, run anywhere" è un obiettivo importante per molte applicazioni e anche se di base i linguaggi assicurano che questa cosa possa essere vera, ogni programmatore ci mette del suo per fare in modo che non lo sia.
- Flessibilità: Molte aziende utilizzano ambienti misti e la capacità di adattarsi è un vantaggio competitivo.
- Comprensione delle differenze: Lavorare su diversi sistemi aiuta a comprendere meglio le peculiarità di ciascuno, migliorando la capacità di debug e ottimizzazione.

Invece di legarsi dogmaticamente a un singolo sistema operativo, un approccio

più produttivo è quello di scegliere lo strumento giusto per il lavoro giusto, mantenendo la flessibilità di muoversi tra diverse piattaforme quando necessario.

4. Il debugger è per i deboli

Sfatiamo un'idea

Un "vero programmatore" scrive codice perfetto al primo tentativo

Questa storiella gira nel mondo della programmazione da anni. Più o meno tutti i programmati famosi si sono cuciti addosso questa immagine, e ognuno di noi è pronto a raccontarla a fine serata quanto le birre sono ormai finite.

La realtà è che il debug e i test sono una parte essenziale ed inevitabile del processo di sviluppo software e questa mentalità di resistenza all'uso di strumenti di debug, vedendoli come una sorta di "stampella", è qualcosa che va superato.

Ormai non conto più le volte che ho preso in mano un software considerato "concluso", pieno i test funzionanti, e piano piano sono usciti fuori problemi, casi d'uso non coperti, problemi di analisi e progettazione: no, l'illusione che la prima stesura di un programma sia in grado di generare un software perfetto e che i test bastino per capire cosa funziona o non funziona è solo un'illusione.

In queste situazioni: log, debug e nuovi test diventano necessari per capire cosa non funziona e come risolverlo.

5. Il codice è tutto nella mia testa

L'analisi, la condivisione della conoscenza e la documentazione sono aspetti spesso trascurati nella programmazione.

Ho tutto in testa

Non esiste una frase meno produttiva di questa, che atrofizza qualsiasi tipo di discussione e di collaborazione.

Esiste una narrativa romantica del programmatore come genio solitario, capace di tenere interi sistemi complessi nella propria mente e di vedere il proprio software come Neo vedeva Matrix.

Questo approccio ha un errore di base: la mente umana, seppur meravigliosa, ha limiti nella quantità di informazioni che può mantenere.

Non tutti siamo Dennis Nedry, il programmatore di Jurassic Park che sapeva tutto il codice del parco a memoria e, anche se lo fossimo, ricordiamoci che fine ha fatto.

Dimenticare il passato è una forma di protezione che viene attivata dal nostro cervello per evitare di impazzire. Questo è il motivo per cui è importante documentare il codice, condividere la conoscenza e lavorare in team.

Ma anche se la propria mente avesse una capacità infinita, ci sono altri motivi per cui il "codice nella mia testa" è un problema:

- Collaborazione: Se il funzionamento del codice è chiaro solo nella mente di chi l'ha scritto, diventa difficile per altri contribuire o mantenere il progetto.
- Propensione agli errori: Senza una documentazione chiara o commenti nel codice, è facile dimenticare dettagli importanti o fare assunzioni errate.
- Onboarding complicato: Nuovi membri del team avranno difficoltà a comprendere e contribuire al progetto.

Occorre quindi che il "celodurista" si renda conto che il codice è un prodotto collaborativo e non condividere delle informazioni non è il miglior approccio per mantenere il proprio lavoro, quanto il modo migliore per perderlo.

Spingiamo quindi questi programmatori a documentare, scrivere Wiki, fare code review, usare diagrammi e schemi, in modo da condividere la conoscenza.

Un approccio che valorizza la documentazione e la condivisione della conoscenza non solo rende il progetto più robusto e mantenibile, ma contribuisce anche alla crescita professionale di tutto il team.

6. I programmatori non usano ChatGPT

Diciamocelo: le intelligenze artificiali sono tutto, fuorché intelligenti.

Guarda quanti errori fa ChatGPT, non può sostituire un programmatore

Le allucinazioni, ma soprattutto l'uso scorretto e superficiale delle AI, porta molti programmatori a pensare che siano strumenti non utilizzabili.

No, le AI non sono motori di ricerca, sono strumenti di aggregazione linguistica, che possono apprendere il nostro contesto di lavoro, e come tali vanno usate.

A queste teoria si sovrappongono sempre di più le teorie che in futuro le AI sostituiranno i programmatori.

Il programmatore "duro e crudo" non usa quindi questi strumenti, perché non ne ha bisogno, perché non funzionano, perché "io sono meglio".

Tutto vero, ma solo in parte. La realtà è che le AI sono strumenti che possono aiutare i programmatori a scrivere codice più velocemente e con meno errori, ma non basta un ora per arrivare a questo risultato, occorrono settimane di utilizzo per capire al meglio come usare questo strumento ed individuare al volo pregi e difetti.

L'AI deve essere vista come strumento di potenziamento, come l'evoluzione dell'autocompletamento dei moderni IDE, ma in grado di estendere il

completamento a un contesto più ampio e complesso.

Ho visto di recente il film Atlas, non per Jennifer Lopez, come molti di voi saranno propensi a pensare, ma per arricchire la mia testa di nuove immagini su futuri possibili. All'interno di questo film l'AI viene vista come un completamento del lavoro dell'uomo ed entrambi si migliorano e potenziano, lavorando in simbiosi.

Per chi ha passato i pomeriggi della propria infanzia a guardare Star Trek: è l'evoluzione della fusione mentale dei vulcaniani o del simbionte dei Trill.

Ci sono molti aspetti della vita di un programmatore che traggono vantaggio da una AI

- Generazione di boilerplate: L'AI può rapidamente produrre strutture di codice di base, permettendo ai programmatori di concentrarsi sugli aspetti più complessi e creativi.
- Debugging assistito: Modelli come ChatGPT possono aiutare a identificare errori nel codice e suggerire possibili soluzioni.
- Esplorazione di nuove tecnologie: L'AI può fornire spiegazioni e esempi di utilizzo per framework o librerie con cui il programmatore non ha familiarità.
- Ottimizzazione: Suggerimenti per migliorare l'efficienza o la leggibilità del codice.
- Test: Generazione di test automatici per verificare il corretto funzionamento del proprio lavoro.

Invece di rifiutare categoricamente l'uso delle AI, i programmatori dovrebbero considerarle come strumenti che possono migliorare la loro produttività e la qualità del loro lavoro.

Conclusioni

Il "celodurismo" nella programmazione, seppur radicato nella storia come simbolo di ingegnosità e ottimizzazione, rischia di diventare un freno all'innovazione e all'efficienza nel mondo dello sviluppo software moderno.

Un approccio più equilibrato riconosce il valore della tradizione e dell'esperienza, ma rimane aperto alle nuove tecnologie e metodologie che possono migliorare il processo di sviluppo.

Il vero segno di un programmatore esperto non è l'adesione dogmatica a pratiche del passato, ma la capacità di valutare criticamente e adottare gli strumenti e le pratiche più adatte a ogni situazione specifica.

Ammiro i celoduristi per la loro tenacia, ma sono sicuro che se avessero più coraggio potrebbero trarre grandi vantaggi da una rivalutazione delle loro posizioni.

I programmatori o mercenari? L'evoluzione del lavoro nel settore IT



Un tempo, costruire un prodotto con un team di programmati era sinonimo di stabilità a lungo termine. Le variazioni nel personale erano rare, limitate a eventi come pensionamenti, trasferimenti personali, burnout e, occasionalmente, offerte di lavoro irresistibili. Ma il vento del cambiamento ha soffiato forte: la diffusione di Internet e la recente pandemia hanno accelerato un processo che sembrava inevitabile: l'avvento del lavoro remoto.

Oggi, persino chi vive in un casolare isolato sulle Alpi può collaborare con un'azienda in una metropoli lontana con la stessa facilità con cui potrebbe lavorare per l'impresa del paese vicino. Il mondo del lavoro IT si è trasformato, e con esso, le regole del gioco.

Da centralizzati a decentralizzati: un nuovo paradigma

Siamo passati da un modello di forte localizzazione di aziende e personale a un concetto rivoluzionario di lavoro distribuito sul territorio. Oggi, parlare di aziende "full remote" non è più un'eresia, ma una realtà consolidata. Il personale lavora da casa o da luoghi diversi, spesso in paesi lontani tra loro. Questa trasformazione ha spalancato le porte a un numero crescente di professionisti, compresi quelli meno inclini al cambiamento, che ora si trovano sommersi da un flusso continuo di offerte lavorative, impensabile solo pochi anni fa.

Nell'era digitale, persino le liste di aziende che offrono lavoro remoto sono diventate una realtà. In Italia, ad esempio, il progetto GitHub "Awesome Italia Remote" raccoglie le aziende italiane che offrono lavori Full Remote, completo di tecnologie richieste e pagine per le candidature:

<https://github.com/italiaremote/awesome-italia-remote>

Questa metamorfosi del lavoro riflette ciò che l'economista Richard Baldwin ha definito "la grande convergenza". Nel suo libro "The Great Convergence: Information Technology and the New Globalization", Baldwin afferma: "L'impatto della tecnologia dell'informazione sta creando una nuova onda di globalizzazione

che permette ai servizi di essere forniti a distanza, cambiando radicalmente il panorama del lavoro globale."

Questa nuova realtà è un terreno fertile di opportunità e sfide, tanto per i lavoratori quanto per le aziende. Per i professionisti dell'IT appassionati del proprio lavoro, liberi da forti legami aziendali e in cerca di salari più alti o ambienti lavorativi stimolanti, questa potrebbe essere considerata un'epoca d'oro. La flessibilità e le opportunità sono cresciute esponenzialmente, aprendo orizzonti prima inimmaginabili.

Dagli anni '80 ad oggi: un viaggio nel tempo del lavoro IT

Ricordo con nostalgia il mio primo impiego in un'azienda di software. Il team era un gruppo affiatato di persone che lavoravano insieme da anni, tutte residenti a pochi chilometri di distanza. In quel contesto, la stabilità sembrava un dato di fatto, e le offerte di lavoro esterne erano rare e spesso poco allettanti.

Quel "paradiso" aziendale, dove non era necessario alcuno sforzo per trattenere i dipendenti, è ormai un ricordo sbiadito.

Oggi, le aziende si trovano di fronte a una sfida titanica: mantenere i propri talenti. La retention del personale è diventata una missione aziendale cruciale. Sostituire un membro del team non è solo dispendioso in termini economici, ma richiede anche un investimento di tempo prezioso per l'addestramento e l'integrazione.

Nel panorama attuale, un programmatore non è più solo un "code monkey". È un professionista poliedrico che concentra su di sé una costellazione di competenze che vanno ben oltre la mera capacità di scrivere codice: conoscenze di prodotto, comprensione delle dinamiche aziendali, empatia con gli utenti finali del software che sviluppa.

I software moderni richiedono professionisti con competenze sempre più specializzate, sia tecnologiche che di prodotto. E sono soprattutto queste ultime a

richiedere tempo per essere affinate e perfezionate.

Come osserva acutamente l'economista del lavoro David Autor del MIT: "Le aziende stanno investendo sempre più in capitale umano specifico dell'impresa, rendendo i lavoratori più produttivi nei loro ruoli attuali".

La specializzazione è un'arma a doppio taglio: da un lato rappresenta un valore aziendale inestimabile, dall'altro può trasformarsi in un rischio per la mobilità del personale. La concentrazione di competenze in poche menti brillanti può rendere l'azienda vulnerabile alla perdita di conoscenze fondamentali in caso di dimissioni o pensionamenti.

Per affrontare questa sfida titanica, molte aziende stanno implementando strategie innovative di retention degne di un romanzo di fantascienza. Queste includono programmi di rimborso delle spese di formazione, allettanti offerte di stock option, revisioni salariali regolari, opportunità di mobilità interna e assegnazioni a breve termine che sembrano missioni spaziali. Inoltre, stanno investendo nella formazione continua, sia per le competenze tecniche che per le soft skill, offrendo maggiore flessibilità nelle modalità lavorative e la possibilità di sperimentare con nuove tecnologie come se fossero esploratori in terre sconosciute.

Fino a qualche anno fa tutte queste opportunità erano riservate a pochi eletti, a professionisti di alto livello o a manager di lungo corso. Oggi, sono diventate la norma, un must have per qualsiasi azienda che voglia rimanere competitiva in un mercato del lavoro sempre più globalizzato e competitivo.

Nel regno dell'IT, la formazione è la linfa vitale degli sviluppatori. Trovare aziende che investono sulla formazione, sia internamente che offrendo budget per la crescita individuale, è come scoprire un'oasi nel deserto. Rappresenta un valore aggiunto inestimabile per la crescita professionale e personale, un motivo per rimanere ancorati all'azienda invece di lasciarsi tentare da sirene che non garantiscono pari opportunità di crescita.

Ma attenzione: il mercato del lavoro IT non è un monolite uniforme. Esistono disparità geografiche abissali: in alcune aree, la formazione è un miraggio e le aziende preferiscono esternalizzare le competenze piuttosto che coltivarle in casa. La competizione è diventata un'arena globale, dove le aziende locali si trovano a combattere non solo tra loro, ma anche contro i giganti tech che sembrano usciti da un film di fantascienza.

La disparità fra grandi e piccole aziende: un abisso digitale

Quando si parla di disparità, il pensiero corre subito alle differenze geografiche, dettate dal territorio in cui le persone lavorano. Ma nel mondo IT, dove la geografia sta perdendo significato, la vera disparità è data da ciò che le aziende possono offrire. E le offerte delle grandi aziende sono sempre più allettanti rispetto a quelle delle piccole realtà.

Questa disparità crea un divario salariale che, all'interno di una zona geografica, può essere mitigato, ma che a livello globale diventa un canyon. Questo abisso salariale è una spada di Damocle per le piccole aziende che cercano disperatamente di trattenere i migliori talenti.

Possiamo quindi parlare di una vera e propria "migrazione digitale": le persone rimangono fisicamente nel loro territorio, ma la loro mente e le loro competenze viaggiano attraverso la rete, lavorando per aziende collocate in qualsiasi angolo del globo. Questo brain drain digitale rappresenta una sfida titanica per le aziende che cercano di trattenere i migliori talenti: creare delle ragioni per trattenere le persone diventa un'impresa degna di Ercole.

Come affrontare queste sfide? Strategie per un nuovo mondo del lavoro

Affrontare in modo corretto la sfida della gestione della conoscenza è un passaggio fondamentale per qualsiasi azienda che voglia sopravvivere in questa giungla digitale. È cruciale evitare di concentrare competenze fondamentali in una

singola persona, come se fosse un oracolo insostituibile. Bisogna invece puntare a creare progetti autonomi e auto-sufficienti, come ecosistemi in grado di prosperare indipendentemente. La perdita di un tecnico esperto può essere un colpo devastante, portando a una significativa riduzione del capitale aziendale, non solo in termini di competenze tecniche, ma anche di conoscenza dei processi e della cultura aziendale.

Dal punto di vista dei professionisti, è importante considerare che cambiare lavoro solo per motivi economici potrebbe non essere sempre la scelta più saggia, soprattutto per i giovani all'inizio della loro odissea professionale. Passare attraverso diverse posizioni e situazioni può fornire un'esperienza preziosa e una maturità inestimabile, ma è essenziale trovare un equilibrio. I soldi sono importanti, certo, ma non sono l'unico tesoro da cercare quando si valuta un'opportunità di lavoro.

Durante i miei primi anni di lavoro, ricordo una frase che mi colpì come un fulmine a ciel sereno:

non meno di 2, non più di 5

Ho poi risentito questo mantra in mille contesti diversi, ma quella volta mi scosse particolarmente. La frase si riferiva al numero di anni che un professionista dovrebbe trascorrere in un'azienda prima di cambiare lavoro. Meno di due anni potrebbe essere interpretato come mancanza di stabilità, più di cinque come mancanza di ambizione. Questo concetto, che può sembrare un fossile in un mondo in rapida evoluzione, ha ancora una certa validità, soprattutto in un settore come l'IT, dove la velocità del cambiamento è paragonabile a quella della luce.

C'è anche un altro aspetto, spesso sottovalutato, che è importante apprendere quando si decide di intraprendere la vita da programmatore: qualsiasi progetto che si affronta, nei primi mesi di sviluppo, sembra un giardino dell'Eden. Problemi? Rari. E anche quando si presentano, si possono superare con relativa facilità, come saltare una pozzanghera.

Ma superati i primi mesi, i progetti iniziano a crescere come giovani titani, diventando sempre più grandi e articolati. Le richieste, sia dei clienti che del management, si trasformano in montagne da scalare.

Col passare del tempo, dare risposte consistenti e funzionali alle richieste diventa una sfida degna di Sisifo. È in questo momento che emergono i veri programmatori, quelli in grado di far funzionare un prodotto stabilmente utilizzato, tenendo conto di una miriade di aspetti che non emergono mai in fase di analisi e durante le prime versioni.

Raggiungere quel livello significa essere diventati dei veri professionisti, capaci di affrontare qualsiasi tipo di sfida e di risolvere qualsiasi tipo di problema. Cambiare progetti ogni sei mesi può aumentare le proprie conoscenze orizzontali, ma a discapito delle conoscenze verticali, quelle che spesso fanno la differenza tra realizzare un prodotto software e farlo funzionare.

Conclusioni: navigare nel nuovo mondo del lavoro IT

In conclusione, siamo entrati in un'era in cui i team sono fluidi come l'acqua e la stabilità è diventata un concetto del passato, un reperto da museo. Le aziende devono adattarsi a questa nuova realtà, offrendo condizioni competitive e un ambiente di lavoro stimolante per attrarre e trattenere i talenti. Allo stesso tempo, i professionisti devono valutare attentamente le opportunità, considerando non solo l'aspetto economico ma anche la crescita professionale e personale, come esploratori alla ricerca del Santo Graal.

La tecnologia sta cambiando la natura del lavoro più velocemente di quanto molte organizzazioni possano adattarsi, come un treno ad alta velocità che sfreccia mentre le stazioni cercano disperatamente di tenergli il passo.

Allo stesso tempo, i programmatore devono fare attenzione a non farsi ingannare dalle facili ricollocazioni aziendali e dal facile guadagno, perché rischiano di dare l'impressione di essere dei semplici mercenari del codice e non delle persone appassionate e competenti nel proprio lavoro. Il vero valore di un programmatore non si misura solo in linee di codice o in stipendio, ma nella capacità di creare, innovare e lasciare un'impronta duratura nel mondo digitale.

In questo nuovo Far West tecnologico, solo chi saprà bilanciare ambizione e fedeltà, competenze tecniche e soft skill, potrà emergere come vero pioniere del coding. Il futuro appartiene a coloro che sapranno navigare queste acque tumultuose con saggezza, adattabilità e una passione incrollabile per il proprio mestiere.

Una RAL migliore non basta per motivare un cambiamento



Periodicamente dedico del tempo a fare dei bilanci su quanto ho realizzato, a riflettere se il mio comportamento è stato corretto e se avrei potuto migliorare in qualche aspetto.

Capita a tutti di sbagliare, di sottovalutare o sopravvalutare una situazione; riaffrontarla a freddo, con una mente critica, aiuta a trovare delle risposte che "a caldo" spesso non emergono.

Noi stessi cambiamo nel tempo e, con noi, cambiano le nostre priorità, i nostri obiettivi e i nostri valori.

L'analisi di oggi è dedicata a una serie di eventi accaduti a fine 2023, che mi hanno fatto riflettere su quanto sia difficile far cambiare le persone, anche in presenza di una proposta allettante, o almeno tale io la ritenevo.

Di tanto in tanto mi capita di dover integrare il mio team di sviluppo con nuove figure, sia per sostituire chi ha lasciato l'azienda sia per ampliare il team per nuovi progetti.

Lasciare un'azienda è una pratica normale che, come ho già descritto, ogni manager dovrebbe mettere in conto, ma che spesso viene sottovalutata.

Per citare Jane Austen:

Nessuno è mai troppo vecchio per cambiare, perché il cambiamento è la sola costante della vita.

Non dobbiamo quindi allarmarci se un nostro collaboratore decide di cambiare, ma dobbiamo essere pronti a gestire il cambiamento, a volte in modo repentino.

I motivi per "lasciare" sono molteplici. Nel mio caso si tratta di programmatore che hanno lasciato l'azienda per andare in altre realtà, di pensionamenti (sì, succede, non è un miraggio), di ricollocazioni: è sbagliato forzare un programmatore a

restare su un progetto che non sente suo. O riusciamo a motivarlo, o è meglio spostarlo in un altro progetto con stimoli diversi, per non danneggiare lui stesso e, di riflesso, il gruppo nel quale è inserito.

Questa volta dovevo trovare una persona per sostituire un programmatore che andava in pensione: normale amministrazione, gestita con largo anticipo, in modo da non avere disservizi e poter gestire al meglio il passaggio di consegne.

Fantastico, ma dov'è il problema?

Il problema si è verificato durante la fase di selezione dei nuovi programmatori che non hanno accettato la mia proposta. E no: il motivo non era la RAL, che era maggiore e in linea con le richieste dei candidati.

In passato pensavo che la RAL fosse il principale motivo per spingere un programmatore a cambiare azienda, influenzato dalle mille volte che ho letto fosse la prima cosa che le persone valutavano nel momento in cui decidevano di cambiare attività, ma mi sono dovuto ricredere.

Focalizzavo troppo il pensiero sulla RAL, vittima del fatto che è uno dei pochi parametri oggettivi che possiamo valutare e del fatto che in molti forum di programmatori è il parametro più discusso e il primo che si valuta ad ogni cambiamento.

Proviamo però a fare un passo indietro e vediamo il tipo di necessità che dovevo soddisfare.

La ricerca era orientata a un programmatore Java esperto interessato a unirsi al nostro team di sviluppo per un prodotto di backend utilizzato a livello europeo.

Focalizzare subito la richiesta penso sia un aspetto fondamentale quando si approccia per la prima volta un candidato: si risparmia tempo a entrambi e si evitano inutili perdite di tempo.

Molti programmatore odiano lavorare su progetti che non conoscono, che non capiscono, che non li stimolano. Parlando subito del progetto e di quale visibilità permette di avere, si fa capire al candidato quanto il suo lavoro sarà importante e visibile.

Il secondo aspetto che affronto è quello tecnologico.

Lavorare con tecnologie ritenute "obsolete" o "vecchie" è un deterrente per molti programmatore.

"Non voglio lavorare su Java, è un linguaggio vecchio, non mi interessa"

è una frase che ho sentito molte volte. Così come spesso ho sentito dire:

"Che versione di linguaggio Java usate? La 8? No grazie, non mi interessa"

Anche se il linguaggio o le tecnologie utilizzate non sono un problema, poter usare una versione recente del linguaggio o tecnologie che non conoscono è un incentivo per molti programmatore.

Allo stesso modo, far capire che il progetto è in continua evoluzione e che si sta cercando di portare il prodotto su una versione più recente del linguaggio è un ulteriore incentivo, purché non venga disatteso.

Nel mio caso si trattava di un progetto partito in Java 11 con un porting a Java 17 completato, che faceva uso di diverse librerie e framework.

Il punto importante è che volevo far percepire che il progetto era in continua evoluzione e che il lavoro del candidato sarebbe stato importante per il successo del progetto.

Il terzo aspetto che affronto è quello della RAL.

Sapendo che è un aspetto importante, indico già al primo colloquio la RAL che posso offrire, in modo da non creare false aspettative: è un vantaggio per tutti. Il candidato sa già a cosa può aspirare e io so se il candidato è interessato o meno.

Focalizzando subito progetto, obiettivi e compensi, il candidato ha già un quadro completo della situazione e può decidere se continuare il processo di selezione o meno.

Un po' per la chiarezza, un po' per la dimostrazione che lo stack tecnologico era aggiornato, un po' per aver esposto da subito la RAL, ho ricevuto un numero adeguato di candidature e, dopo una fase di selezione, ho individuato alcuni candidati promettenti.

Andavano bene a me, andavano bene al team, andavano bene ai manager e, soprattutto, tutte le richieste del candidato erano soddisfatte: lavorare in un ambito conosciuto, usare tecnologie recenti, avere una RAL maggiore.

Nonostante le premesse che mi incoraggiavano a pensare che tutto si potesse risolvere in modo positivo, ho ricevuto delle risposte negative che non mi aspettavo:

"Non me la sento"

Quando la RAL non basta

Il motivo principale per cui alcuni candidati hanno rifiutato la nostra proposta è stato quello di non sentirsi pronti al cambiamento: il lavoro attuale, pur essendo meno stimolante, meno interessante, meno pagato, era comunque un lavoro sicuro e conosciuto.

Il cambiamento, anche se in meglio, era percepito come un rischio, un salto nel

vuoto.

Questo mi ha fatto ragionare sul fatto che, anche se ti trovi in un'azienda che non ti soddisfa e puoi avere tutti i motivi per cambiare, se non sei incline al cambiamento non cambierai, o quantomeno non lo farai per un cambiamento minimo. Ci vogliono delle motivazioni più forti o una dialettica in grado di far percepire il cambiamento non come un momento di stress, ma come un momento di crescita e valorizzazione.

Questo fenomeno l'ho notato soprattutto tra le figure più mature, che preferiscono rimanere in contesti noti piuttosto che affrontare un cambiamento, anche se benefico.

La "comfort zone", anche se problematica, può farci accettare situazioni tossiche come normali.

Il più grande ostacolo al cambiamento non è l'ignoranza o la resistenza, ma l'illusione che ciò che facciamo ora sia sicuro.

Questo mi ha fatto anche pensare a tutte le volte che qualcuno mi ha confessato di essere insoddisfatto del proprio lavoro, ma di non voler cambiare per paura di non trovare di meglio.

La paura del cambiamento è un sentimento comune, ma può essere superata con le giuste strategie.

In base a quanto ho potuto osservare, alcune strategie che possono aiutare a superare la paura del cambiamento sono:

- Piccoli Passi: per chi non ama il cambiamento "brutale", la tecnica dei piccoli passi può aiutare. Senti che quello che stai facendo non ti soddisfa? Inizia a studiare quello che vorresti fare. Ogni giorno dedica del tempo per farlo e vedrai che col tempo avrai abbastanza conoscenze per fare il salto che tanto desideravi.

- Mentalità di Crescita: non guardare le sfide con paura, ma come opportunità per crescere ed evolvere la tua situazione. Avere una mentalità orientata alla crescita permette di uscire in modo quasi trasparente dalla propria gabbia mentale.
- Visualizzazione: non guardare con negatività il cambiamento, ma prova a portare la mente oltre e a pensare a cosa potrebbe accadere di positivo una volta cambiato.
- Supporto Sociale: cercare il sostegno di colleghi, amici o mentori aiuta a capire meglio cosa può accadere e può fornire l'incoraggiamento necessario a superare l'ostacolo mentale che impedisce di cambiare.

Gli errori che ho commesso

È facile dare la colpa ai candidati se qualcuno decide di non accettare la tua offerta. Affrontare l'argomento in modo intelligente dovrebbe invece far porre la domanda: dove ho sbagliato?

Se non siamo in grado di capire cosa non ha funzionato, non saremo in grado di migliorare.

Nel mio caso ho provato a cercare la risposta oltre la RAL, oltre il progetto, oltre le tecnologie utilizzate.

E se dietro quel "non me la sento" in realtà ci fosse un "non è esattamente quello che cerco"?

Ho cercato di capire cosa avrebbe potuto spingere un programmatore a non accettare una proposta che sembrava allettante.

Una delle ragioni che mi sono dato era sicuramente il contesto aziendale nel quale il candidato avrebbe dovuto lavorare, o almeno il modo in cui l'avevo dipinto durante il colloquio.

Qualcuno parla di "corporate appeal", ovvero l'attrattiva che un'azienda ha nei confronti di un candidato.

Il corporate appeal è un insieme di fattori che vanno dallo stile di lavoro alla cultura aziendale, dai benefit offerti alla possibilità di crescita, dalla possibilità di formazione alla possibilità di lavorare in un ambiente stimolante.

Il corporate appeal è un fattore importante che spesso viene sottovalutato, ma che può fare la differenza tra un candidato che accetta la proposta e uno che la rifiuta.

Mostrare che la propria azienda ha dei valori, crede nella valorizzazione delle persone, offre la possibilità di crescere e di formarsi, offre un ambiente di lavoro stimolante e confortevole, può fare la differenza tra un candidato che accetta la proposta e uno che la rifiuta.

Conclusioni

Pensare che solo offrendo una RAL maggiore si possa motivare un cambiamento è un errore che ho commesso nel corso della mia vita.

Non ho portato a bordo persone che avrebbero potuto fare la differenza per il mio team e per il mio progetto.

Questo mi ha insegnato che il lavoro maggiore che dovrebbe fare ogni manager è quello di rendere la propria azienda attraente per i candidati, di far percepire che lavorare in quell'azienda è un'opportunità di crescita e di valorizzazione.

Far percepire al candidato che potrà lavorare in autonomia, con i giusti strumenti, che crescerà di anno in anno e sarà una pedina importante per il successo dell'azienda.

A quel punto passano in secondo piano molti aspetti che prima erano considerati fondamentali.

La RAL, il progetto, le tecnologie utilizzate diventano solo un dettaglio, un aspetto secondario rispetto al corporate appeal e alla possibilità di crescita che l'azienda offre.

Chi perde un programmatore perde un tesoro



Un vecchio detto recita: "Chi trova un amico trova un tesoro". Il significato positivo di questa frase risiede nel fatto che l'amicizia, quella vera, quella che ti fa dormire tranquillo perché sai che qualsiasi cosa succederà avrai sempre un amico su cui contare, che ti offre aiuto per volere e non per costrizione, è una cosa rara.

Allo stesso modo, un buon programmatore è un tesoro per un'azienda. È una persona che, grazie alle sue competenze tecniche, alla sua esperienza e alla sua capacità di problem solving, può fare la differenza tra il successo e il fallimento di un progetto.

Se volessimo però guardare al rovescio della medaglia, "Chi perde un programmatore perde un tesoro". Troppo spesso si sottovaluta la perdita di qualcuno che nel tempo ha accumulato esperienza e competenze all'interno dell'azienda e ora decide di andare altrove.

Troppe volte ho visto aziende avere dei contraccolpi a causa di una persona che lasciava l'azienda, che fosse un programmatore, un project manager o un analista. Questo articolo vuole essere una riflessione su cosa significa perdere un programmatore e su come le aziende possano evitare che ciò accada.

La vita professionale di un programmatore

I programmatori, come tutti i professionisti, attraversano diverse fasi significative durante la loro carriera lavorativa. Spesso però è possibile individuare un pattern ripetitivo che si manifesta con particolare frequenza in questo settore.

La prima fase è quella dell'inserimento in un progetto. Si tratta di un periodo iniziale, più o meno lungo, dove la persona viene introdotta nel contesto lavorativo. È un momento caratterizzato da studio intenso alternato a momenti di lavoro pratico, spesso sotto la supervisione di un programmatore più esperto che funge da guida.

Segue poi l'inizio delle attività lavorative vere e proprie. In questa fase il

programmatore inizia a lavorare sui primi task assegnati, prende confidenza con il progetto e con le persone del team. Non si è ancora pienamente produttivi, ma si inizia a comprendere le dinamiche del progetto e del gruppo di lavoro.

La terza fase è quella della maturità professionale. Il programmatore inizia a lavorare in completa autonomia, sa con chi deve interfacciarsi, conosce a fondo i processi aziendali, comprende le problematiche e sviluppa soluzioni efficaci. È un momento di grande crescita e soddisfazione professionale.

Infine arriva la fase di assestamento, dove il programmatore ha ormai in mano quasi tutto il codice del progetto. L'emozione e la curiosità che caratterizzavano le fasi iniziali tendono gradualmente a diminuire. Si passa da una situazione in cui "tutto è possibile", pervasa da euforia e curiosità, ad una fase di maggiore consapevolezza, fino ad arrivare talvolta a pensare che "tutto è già stato fatto" o che "nulla si può più fare".

È proprio in questo momento che il programmatore inizia a interrogarsi, ponendosi giustamente delle domande sul proprio futuro professionale, sul lavoro attuale, sul progetto e sull'azienda.

Come dice Aristotele nel libro "Etica Nicomachea":

La felicità non è un momento o uno stato che si raggiunge una volta per tutte, ma un processo che dura tutta la vita e richiede costante impegno nell'esercizio della virtù.

Per questo motivo le aziende dovrebbero intervenire in modo proattivo per supportare i propri programmatori in questa fase.

La perdita di stimoli

Questa fase rappresenta uno dei momenti più critici della vita lavorativa: la graduale perdita di stimoli, ambizioni, prospettive e obiettivi. Tutto appare grigio, o

comunque cristallizzato in una situazione che sembra immutabile.

Ci sono professionisti che affrontano questa fase in modo costruttivo, incanalando le proprie energie nel miglioramento del codice esistente, nella migliore strutturazione del progetto, nell'ottimizzazione dei processi, nel perfezionamento della documentazione. Si dedicano a quelle attività del backlog che troppo spesso vengono rimandate e che trovano spazio solo nei momenti di apparente stasi.

Altri invece iniziano a sviluppare un profondo senso di sconforto, di noia, di frustrazione. Cominciano a guardare oltre, alla ricerca di nuovi stimoli, nuovi progetti, nuove realtà aziendali. E dallo sconforto al salto verso una nuova azienda, il passo è sorprendentemente breve.

Questo processo, a ben vedere, non è completamente negativo. In molte professioni si arriva ad un punto di saturazione che spinge al cambiamento. Il vantaggio peculiare del programmatore è che, se possiede le competenze adeguate, può effettuare questo cambiamento con relativa facilità rispetto ad altre professioni. Può trasformare la propria situazione lavorativa in modo rapido e, spesso, senza nemmeno dover abbandonare la propria postazione di lavoro.

La perdita di un programmatore

Contrariamente a quanto si potrebbe superficialmente pensare, un programmatore non rappresenta semplicemente un elemento sostituibile nella catena del valore di un'azienda. Questo è particolarmente vero in contesti dove il team di lavoro non è molto numeroso e le competenze sono altamente specifiche, racchiuse nella mente di poche persone chiave.

Non tutti lavorano in grandi aziende, ben strutturate e ridondate. Spesso ci si trova in piccole aziende, dove il team di sviluppo è sotto la decina di persone e dove la perdita di un programmatore può avere conseguenze devastanti.

A complicare ulteriormente la situazione, ci sono i progetti stessi che non hanno

una documentazione adeguata, sono frutto di anni di lavori sovrapposti e dove i processi non sono strutturati in modo ottimale. In queste situazioni la conoscenza olistica, ove presente, rimane concentrata nelle mani di un numero ristretto di persone.

In questo scenario, perdere un programmatore diventa un problema serio, perché si perde veramente un tesoro e, in alcuni casi, persino il controllo effettivo del progetto che stava seguendo.

Anche nelle realtà aziendali più strutturate, dove i team sono ben organizzati e il codice è accuratamente documentato e testato, emerge la tendenza a ricercare qualcuno che sappia replicare esattamente le competenze e le attività del programmatore uscente.

Questa aspirazione si rivela spesso irrealistica, poiché ogni programmatore possiede il proprio stile distintivo, le proprie abitudini consolidate, le proprie competenze specifiche. Non è affatto semplice trovare qualcuno che possa sostituirlo in modo perfettamente sovrapponibile.

In queste situazioni si sente frequentemente ripetere: "sono mesi che cerco qualcuno, ma non lo trovo". Questa affermazione, apparentemente banale, racchiude in realtà diverse verità significative: evidenzia come la persona da sostituire possedesse competenze specifiche che il team attuale non riesce a coprire e suggerisce che probabilmente il compenso che percepiva era sottodimensionato rispetto al reale valore che apportava all'organizzazione.

Un altro aspetto frequentemente sottovalutato in questi frangenti è l'aspettativa che il nuovo programmatore possa, sin dal primo giorno, coprire totalmente o parzialmente il lavoro del predecessore. Sorprendentemente, c'è ancora chi coltiva questa illusione.

È fondamentale invece mettere in conto che, per un periodo significativo, il team non potrà mantenere gli stessi livelli di performance. Questa problematica diventa

ancora più critica se il programmatore in uscita è uno dei più esperti o se il team è composto da un numero limitato di persone.

Come evitare la perdita di un programmatore

La consapevolezza del valore di un programmatore dovrebbe spingere le aziende a creare un ambiente lavorativo che favorisca la sua permanenza e il suo sviluppo professionale. L'elemento fondamentale in questo senso è garantire un adeguato livello di autonomia nel lavoro quotidiano. Un programmatore dovrebbe avere la libertà di organizzare il proprio tempo e le proprie attività, scegliendo le modalità più efficaci per raggiungere gli obiettivi stabiliti. Questa autonomia non significa isolamento o mancanza di coordinamento con il team, ma piuttosto la possibilità di esprimere al meglio la propria creatività e le proprie competenze tecniche senza eccessive costrizioni procedurali.

L'azienda deve inoltre assicurarsi che il programmatore abbia a disposizione tutti gli strumenti necessari per svolgere al meglio il proprio lavoro. Questo significa non solo fornire hardware e software adeguati, ma anche garantire l'accesso a risorse di formazione, documentazione e supporto tecnico. Troppo spesso le aziende sottovalutano l'impatto negativo che strumenti obsoleti o inadeguati possono avere sulla motivazione e sulla produttività di un programmatore. La frustrazione derivante dal dover lottare quotidianamente con limitazioni tecniche può essere un forte incentivo a cercare opportunità altrove.

Platone all'interno de "La Repubblica, libro IV" diceva, parlando dei pentolai:

Se a causa della povertà non potrà procurarsi gli strumenti o altri utensili indispensabili alla sua arte, realizzerà prodotti più scadenti e renderà artigiani inferiori i figli o altri a cui insegnerà il proprio mestiere

Un aspetto cruciale, spesso sottovalutato, è quello di far sentire il programmatore veramente parte del progetto su cui sta lavorando. Questo va ben oltre il semplice coinvolgimento tecnico: significa renderlo partecipe delle decisioni strategiche,

ascoltare le sue opinioni sulle scelte architettoniche, coinvolgerlo nella pianificazione delle attività future. Un programmatore che si sente veramente parte del progetto sviluppa un senso di appartenenza e responsabilità che va oltre il semplice rapporto professionale. Questo legame emotivo con il progetto e con il team può essere un potente fattore di retention, specialmente nei momenti di difficoltà o quando si presentano opportunità alternative.

Quando un programmatore percepisce di avere voce in capitolo nelle decisioni che riguardano il suo lavoro, di poter contare su strumenti all'altezza delle sfide che deve affrontare e di essere parte integrante di un progetto significativo, le probabilità che cerchi alternative professionali diminuiscono sensibilmente. Non si tratta solo di trattenere un professionista qualificato, ma di creare le condizioni perché possa esprimere al meglio il proprio potenziale, contribuendo così non solo al successo del progetto ma anche alla crescita dell'intero team.

Conclusioni

Il percorso professionale di un programmatore rappresenta un viaggio complesso, caratterizzato da diverse fasi di crescita, momenti di entusiasmo e periodi di riflessione. È importante che le aziende siano consapevoli di questi momenti critici e si impegnino attivamente a creare un ambiente di lavoro che favorisca la permanenza e lo sviluppo dei propri programmatori.

La metafora del tesoro, con cui abbiamo aperto questa riflessione, acquista ancora più significato quando si considera l'insieme di conoscenze, esperienza e valore che un programmatore porta con sé. Non si tratta solo di competenze tecniche, ma di un patrimonio fatto di comprensione profonda dei processi, delle dinamiche di progetto e delle relazioni all'interno del team.

Quando un'azienda perde un programmatore, non perde semplicemente una risorsa, ma un pezzo della propria storia e del proprio futuro.

La sfida per le organizzazioni moderne non è quindi solo quella di attrarre talenti, ma di creare un ambiente in cui questi possano crescere, esprimersi e trovare continui stimoli. Solo attraverso un impegno concreto nel valorizzare i propri programmatori, fornendo loro autonomia, strumenti adeguati e un reale senso di appartenenza, le aziende possono sperare di conservare questo tesoro. In un mondo dove la tecnologia evolve rapidamente e le competenze sono sempre più preziose, investire nella soddisfazione e nella crescita dei propri programmatori non è più solo una scelta, ma una necessità strategica per il successo di lungo termine.

Biografia

Matteo Baccan è un ingegnere del software e formatore professionista con oltre 30 anni di esperienza nel settore IT.

Ha lavorato per diverse aziende e organizzazioni, occupandosi di progettazione, sviluppo, testing e gestione di applicazioni web e desktop, utilizzando vari linguaggi e tecnologie. È anche un appassionato divulgatore e insegnante di informatica, autore di numerosi articoli, libri e corsi online rivolti a tutti i livelli di competenza.

Gestisce un sito internet e un canale YouTube dove condivide video tutorial, interviste, recensioni e consigli sulla programmazione.

Attivo nelle community open source, partecipa regolarmente a eventi e concorsi di programmazione.

Si definisce un "sognatore realista" che ama sperimentare, innovare e condividere le sue conoscenze e passioni, seguendo il motto: "Non smettere mai di imparare, perché la vita non smette mai di insegnare".

Indice

Prefazione	2
Ringraziamento	4
Introduzione	5
"Diventa programmatore dopo i 40 anni"	6
La realtà	7
Bias anagrafici	8
La proposta formativa	9
Conclusioni	13
Riferimenti	13
I programmatori che concludono tutti i task non hanno finito la loro giornata lavorativa	14
Chiudere i task non è l'obiettivo finale	15
La sindrome del cavallo da corsa	16
Premiare l'individualismo	16
Il problema del "finire i task"	17
Di chi è la responsabilità?	17
Quali strategie possiamo mettere in atto per mitigare questo problema?	18
Gestione delle superstar	19
L'effetto "catena di montaggio"	20
Conclusioni	22
La forza di ammettere di non sapere	23
La paura di non sapere	24
Ero ignorante e non lo sapevo	25
L'ignoranza è una risorsa	26
Gestione dell'ignoranza	28
Il coraggio di chiedere	28
Il mito del tuttologo	29
Chiedere da solo però non basta	30

La differenza fra prosciutto cotto e prosciutto crudo	30
La paura del giudizio	31
Fail-forward culture	32
Conclusioni	33
Oltre il Full Stack: Percorsi di Crescita nello Sviluppo Software	34
"Puoi cambiare questa condizione nel codice? Cosa ci vuole?"	36
La paura dei senior	37
I senior non sono cattivi	38
La manutenzione di un progetto cambia le regole?	39
Caso d'uso di una nota Banca italiana	40
Qual è la percezione della modifica all'esterno del progetto?	41
Conclusioni	43
L'adozione di nuovi framework potrebbe far fallire il tuo progetto	44
E i framework?	46
L'orizzonte temporale: una dimensione cruciale	47
Il rischio dell'abbandono	49
L'importanza dell'analisi periodica	50
Il rischio dei servizi non standard	50
Conclusioni	52
Il Mito del Full Stack Developer: una realtà scomoda	53
Il mio approccio col termine Full Stack Developer	54
Il mercato del lavoro e il Full Stack Developer	56
La qualità del codice e la valutazione del programmatore	56
Se lavorassimo invece sulla "problem solving attitude"?	58
Conclusioni	60
"Vediamo chi c'è l'ha più grosso: facciamo code review"	61
La code review da fastidio	62
Il codice non usato	64
Io uso i TAB, chi usa gli spazi non è un programmatore bravo	64

Resistenza al cambiamento	66
Clean Code	66
Automatizzare il possibile	67
Ma a me non interessa	67
Conclusione	70
Il "celodurismo" dei programmatori	71
Le radici storiche	72
Il manifesto del celodurismo	73
1. L'IDE non serve a nulla	73
2. La CLI è la soluzione a tutti i problemi	74
3. Non usiamo Windows perché i programmatori usano Linux (o MacOS)	75
4. Il debugger è per i deboli	77
5. Il codice è tutto nella mia testa	77
6. I programmatori non usano ChatGPT	79
Conclusioni	81
I programmatori o mercenari? L'evoluzione del lavoro nel settore IT	82
Da centralizzati a decentralizzati: un nuovo paradigma	83
Dagli anni '80 ad oggi: un viaggio nel tempo del lavoro IT	84
La disparità fra grandi e piccole aziende: un abisso digitale	86
Come affrontare queste sfide? Strategie per un nuovo mondo del lavoro	86
Conclusioni: navigare nel nuovo mondo del lavoro IT	89
Una RAL migliore non basta per motivare un cambiamento	90
Fantastico, ma dov'è il problema?	92
Quando la RAL non basta	94
Gli errori che ho commesso	96
Conclusioni	98
Chi perde un programmatore perde un tesoro	99
La vita professionale di un programmatore	100
La perdita di stimoli	101
La perdita di un programmatore	102

Come evitare la perdita di un programmatore	104
Conclusioni	106
Biografia	107
Indice	108