Capitolo 1

Essere professionali

"Ahahaha! Ridi, vecchio mio. È un tiro giocato dal Signore, dal destino o dalla natura, come preferisci tu. Ma chiunque o qualsiasi cosa l'abbia giocato aveva molto spirito!"
– Howard, *Il tesoro della Sierra Madre*

In questo capitolo

- **Fate attenzione a ciò che chiedete**
- **Assumersi le responsabilità**
- **Primo, non nuocere**
- **Etica del lavoro**
- **Identificatevi con il vostro datore di lavoro/cliente**
- **Umiltà**
- **Bibliografia**

E allora volete diventare sviluppatori professionali, è così? Volete camminare a testa alta e dichiarare al mondo: "Sono un professionista!". Volete che tutti vi guardino con rispetto e vi trattino con deferenza. Volete vedere madri che vi indicano col dito e vi portino a esempio ai loro figli. Volete tutto questo. Giusto?

Fate attenzione a ciò che chiedete

Il termine "professionalità" è carico di significato. Di certo è un distintivo d'onore e orgoglio, ma è anche un segno di responsabilità e obbliga a dare conto delle scelte. Le due cose vanno di pari passo, ovviamente. Non potete essere orgogliosi e onorati di qualcosa di cui non volete essere ritenuti responsabili.
È molto più facile essere non-professionisti. I non-professionisti non devono assumersi la responsabilità del lavoro che svolgono, lasciano che se ne occupino i loro superiori. Se un non-professionista commette un errore, il datore di lavoro sistema il pasticcio. Quando invece è un professionista a commettere un errore, è lui a dover sistemare il pasticcio.
Che cosa accadrebbe se vi lasciaste sfuggire un bug in un modulo e ciò costasse alla vostra azienda 10.000 dollari? Il non-professionista scrollerebbe le spalle, direbbe "Sono cose che succedono" e inizierebbe a scrivere un altro modulo. Il professionista verserebbe all'azienda un assegno di 10.000 dollari! (Auspicabilmente avrà adottato una buona politica a proposito di errori e omissioni!)
Sì: è un po' diverso quando si tratta dei vostri soldi, vero? Ma quello è l'atteggiamento di un professionista. In effetti, quell'atteggiamento è l'essenza stessa della professionalità. Perché, vedete, la professionalità è tutta una questione di assunzione di responsabilità.

Assumersi le responsabilità

Avete letto l'Introduzione, vero? Se non l'avete fatto, tornate indietro e fatelo ora; definisce il contesto per tutto ciò che leggerete in questo libro.
Ho imparato ad assumermi le mie responsabilità subendo le conseguenze del non assumermele.
Nel 1979 lavoravo per un'azienda, la Teradyne. Ero "ingegnere responsabile" di un software che controllava un sistema basato su mini- e microcomputer che valutava la qualità delle linee telefoniche. Il minicomputer centrale era collegato tramite linee telefoniche dedicate o dial-up a 300 baud a decine di microcomputer satelliti che controllavano l'hardware di misurazione. Il codice era tutto scritto in Assembler.
I nostri clienti erano i responsabili del servizio delle principali compagnie telefoniche. Ognuno di loro aveva la responsabilità di 100.000 linee telefoniche o più. Il mio sistema aiutava questi responsabili di area a trovare e riparare i malfunzionamenti e i problemi nelle linee telefoniche prima che i loro clienti se ne lamentassero. Ciò riduceva i tassi di reclamo dei clienti che le commissioni misuravano e utilizzavano poi per regolare le tariffe che le compagnie telefoniche potevano applicare. In breve, questi sistemi erano assolutamente importanti.
Ogni notte, questi sistemi avviavano una "routine notturna" in cui il minicomputer centrale ordinava a ciascuno dei microcomputer satelliti di sottoporre a test ogni linea telefonica che era sotto il loro controllo. Ogni mattina, il computer centrale produceva

l'elenco delle linee difettose, insieme alle caratteristiche del difetto. I responsabili dell'area di servizio utilizzavano questo rapporto per inviare il personale per riparare i guasti prima che i clienti potessero lamentarsene.

In un'occasione ho spedito una nuova release del sistema a diverse decine di clienti. "Spedire" è esattamente la parola giusta. Ho scritto il software su nastri e ho spedito quei nastri ai clienti. I clienti hanno caricato i nastri e hanno fatto il reboot dei sistemi. La nuova release correggeva alcuni piccoli difetti e aggiungeva una nuova funzionalità che i nostri clienti avevano richiesto. Avevamo detto loro che avremmo fornito quella nuova funzionalità entro una determinata data. Sono riuscito per un soffio a spedire i nastri quella sera stessa, in modo che arrivassero entro la data promessa.

Due giorni dopo ho ricevuto una chiamata dal nostro responsabile dell'assistenza sul campo, Tom. Mi ha detto che diversi clienti si erano lamentati del fatto che la "routine notturna" non era stata completata e che non avevano ricevuto alcun report. Mi sono sentito mancare, perché, per spedire il software in tempo, avevo trascurato di sottoporre a test la routine. Avevo provato molte altre funzionalità del sistema, ma il test della routine avrebbe richiesto ore e io dovevo spedire il software subito. Nessuna delle correzioni di bug riguardava il codice della routine, quindi mi sentivo al sicuro.

Perdere un report notturno era un *grosso problema*. Significava che i riparatori non avrebbero avuto lavoro da svolgere ma sarebbero stati sovraccaricati in seguito. Significava che alcuni clienti avrebbero potuto notare un guasto e lamentarsene. Perdere i dati di una notte è sufficiente a far sì che un responsabile dell'area di servizio chiami Tom con un tono non esattamente gentile.

Ho avviato il nostro sistema di laboratorio, ho caricato il nuovo software e ho iniziato una routine. Ci sono volute diverse ore, ma poi si è bloccata. La routine falliva. Se avessi eseguito questo test prima di spedire la release, le aree di servizio non avrebbero saltato la routine serale e i responsabili delle aree di servizio ora non starebbero assillando Tom. Ho telefonato a Tom per dirgli che potevo replicare il problema. Mi ha detto che la maggior parte degli altri clienti lo aveva chiamato con lo stesso reclamo. Poi mi ha chiesto entro quanto avrei potuto risolvere il problema. Gli ho detto che non lo sapevo, ma che ci stavo lavorando. Nel frattempo, ho suggerito che i clienti tornassero alla precedente release del software. Si è arrabbiato con me, perché gli stavo comunicando un doppio colpo per i clienti: non solo avevano perso i dati di un'intera notte, ma non potevano nemmeno usare la nuova funzionalità che era stata loro promessa.

Il bug è stato difficile da trovare e i test hanno richiesto diverse ore. La prima correzione non ha funzionato. E nemmeno la seconda. Ci sono voluti diversi tentativi, e diversi giorni, per capire che cosa fosse andato storto. In tutto questo tempo, Tom mi chiamava ogni tot ore chiedendomi quando avrei finito di sistemare il problema. Si è anche assicurato che fossi a conoscenza delle lamentele che stava ricevendo dai responsabili delle aree di servizio e di quanto fosse imbarazzante per lui dire loro di rimettere su i vecchi nastri.

Alla fine, ho trovato il difetto, ho spedito i nuovi nastri e tutto è tornato alla normalità. Tom, che non era il mio capo, si è calmato e ci siamo lasciati l'intero episodio alle spalle. Poi, quando tutto è finito, il mio capo è venuto da me e mi ha detto: "Scommetto che non lo farai più". Potevo solo annuire.

Riflettendoci, mi sono reso conto che spedire la routine senza sottoporla a test era stato da irresponsabili. Il motivo per cui ho trascurato il test era per poter dire di aver spedito il nastro entro la scadenza. Volevo salvare la faccia. Non mi sono preoccupato del cliente, né del mio datore di lavoro, ma solo della mia reputazione. Avrei dovuto assumermi

subito la responsabilità e dire a Tom che senza i test finali non ero pronto per spedire il software in tempo. Sarebbe stato difficile per me e Tom si sarebbe arrabbiato, ma nessun cliente avrebbe perso dati e nessun responsabile del servizio ci avrebbe chiamato.

Primo, non nuocere

E allora, come ci assumiamo le nostre responsabilità? Ci sono alcuni principi. Attingere al giuramento di Ippocrate può sembrare fin troppo arrogante, ma quale fonte migliore? E, in effetti, non ha forse perfettamente senso che la prima responsabilità, e il primo obiettivo, di un aspirante professionista sia quello di usare i propri poteri a fin di bene? Quale danno può fare uno sviluppatore di software? Da un punto di vista puramente software, può fare danni sia alla funzionalità sia alla struttura del software. Vediamo come evitarlo.

Non danneggiare la funzionalità

Chiaramente, vogliamo che il nostro software funzioni. In effetti, la maggior parte di noi, oggi, fa il programmatore perché ha dovuto far funzionare qualcosa una volta e vuole provare di nuovo quella sensazione. Ma non siamo gli unici a volere che il software funzioni. Anche i nostri clienti e datori di lavoro vogliono la stessa cosa. E, infatti, ci pagano proprio per creare software che funzioni esattamente come vogliono loro.

Noi danneggiamo la funzionalità del nostro software quando creiamo bug. Pertanto, per essere professionali, non dobbiamo creare bug.

"Ma… aspetta!", vi sento dire: "Non è ragionevole. Il software è troppo complesso per essere esente da bug".

Certo, avete ragione. Il software è troppo complesso per essere esente da bug. Ma sfortunatamente questo non vi tira fuori dai guai. Il corpo umano è troppo complesso per essere compreso nella sua interezza, ma i medici prestano comunque giuramento di non farci mai del male. E se vale per loro, *come può non valere per noi*?

Vi sento obiettare: "Ci stai dicendo che dobbiamo essere perfetti?".

No: vi sto dicendo che dovete essere responsabili delle vostre imperfezioni. Il fatto che nel vostro codice siano presenti sicuramente dei bug non significa che non ne siate responsabili. Il fatto che il compito di scrivere un software perfetto sia virtualmente impossibile non significa che non siate responsabili delle imperfezioni.

È compito di un professionista essere responsabile degli errori, anche se sono praticamente certi. Quindi, signori aspiranti professionisti, la prima cosa che dovete fare è scusarvi. Le scuse sono necessarie, ma non sufficienti. Non potete semplicemente continuare a commettere gli stessi errori più e più volte. Man mano che maturate nella vostra professione, il vostro tasso di errori dovrebbe rapidamente diminuire verso l'asintoto zero. Non arriverà mai a zero, ma è vostra responsabilità avvicinarvi il più possibile a questo zero.

Il controllo qualità non dovrebbe mai trovare nulla

Pertanto, quando rilasciate il vostro software dovete aspettarvi che il controllo qualità non trovi problemi. È estremamente poco professionale inviare intenzionalmente al controllo

qualità del codice che sapete essere difettoso. E quale codice sapete essere difettoso? Qualsiasi codice di cui non siete *certi*!
Alcuni usano il controllo qualità come un "cacciatore di bug". Inviano loro del codice che non hanno controllato a fondo. Si affidano al controllo qualità perché trovi i bug e li segnali agli sviluppatori. In effetti, alcune aziende premiano il controllo qualità in base al numero di bug che trovano. Più bug ci sono, maggiore è la ricompensa.
Non importa che questo sia un comportamento terribilmente costoso che danneggia l'azienda e il software. Non importa che questo comportamento rovini i programmi e mini la fiducia dell'azienda nel team di sviluppo. Non importa che questo comportamento sia semplicemente da pigri e irresponsabili. Rilasciare al controllo qualità del codice che non sapete se funziona è poco professionale. Viola la regola del "non nuocere".
Il controllo qualità troverà dei bug? Probabilmente sì, quindi preparatevi a scusarvi, e poi scoprite perché quei bug sono riusciti a sfuggirvi e fate qualcosa per impedire che la cosa accada di nuovo.
Ogni volta che il controllo qualità o, peggio, un utente riscontra un problema, dovete essere sorpresi, dispiaciuti ma anche determinati a impedire che la cosa si ripeta.

Dovete sapere che funziona

Come potete *sapere* se il vostro codice funziona? È facile. Sottoponetelo a test. E poi di nuovo. Provatelo in lungo e in largo. Provatelo in mille modi diversi!
Forse temete che sottoporre a test così tanto il vostro codice vi richiederà troppo tempo. Dopotutto avete programmi e scadenze da rispettare. Se trascorrete tutto il vostro tempo a sottoporre a test, non scriverete mai nient'altro. Giusto! E allora automatizzate i vostri test. Scrivete unit test che possiate eseguire all'ultimo minuto ed eseguiteli il più frequentemente possibile.
Quanto del codice dovrebbe essere sottoposto a test con questi unit test automatizzati? Devo davvero rispondere a questa domanda? Tutto! Tutto quanto.
Sto suggerendo di adottare una copertura dei test pari al 100%? No, non lo sto solo *suggerendo*. Lo sto proprio *chiedendo*. Ogni singola riga di codice che scrivete dovrebbe essere sottoposta a test. Punto.
Sembra irrealistico? Ovviamente no. Scrivete codice perché vi aspettate che funzioni. E se vi aspettate che funzioni, dovete sapere che funziona. E l'unico modo per saperlo è sottoporlo a test.
Sono il principale contributore e committer di un progetto open source chiamato *FitNesse*. Al momento in cui scrivo, FitNesse è costituito da 60.000 righe di codice. 26 di questi sono scritti in più di duemila unit test. Emma segnala che la copertura di quei duemila test è di circa il 90%.
Perché la copertura del codice non è più alta? Perché Emma non riesce a vedere tutte le righe di codice che vengono eseguite! Credo che la copertura sia molto più alta di così. E allora la copertura è del 100%? No, il 100% è un asintoto.
Ma una certa parte del codice non è difficile da sottoporre a test? Sì, ma solo perché quel codice è stato *progettato senza considerare i test*. La soluzione consiste nel progettare il vostro codice in modo che sia *facile da sottoporre a test*. E il modo migliore per farlo è scrivere prima i test e solo dopo il codice che li deve superare.
Si tratta di una disciplina nota come TDD (*Test Driven Development*), di cui parleremo più approfonditamente in un prossimo capitolo.

Controllo qualità automatizzato

L'intera procedura del controllo qualità per FitNesse è l'esecuzione dei test: unit test e test di accettazione. Se questi test vengono superati, spedisco. Ciò significa che la mia procedura di controllo qualità richiede circa tre minuti e posso eseguirla quando e quanto voglio.

Ora, è vero che nessuno muore se c'è un bug in FitNesse. Nessuno perde milioni di dollari. D'altro canto, FitNesse ha molte migliaia di utenti e una lista di bug molto corta. Certamente, alcuni sistemi sono così *mission-critical* che un breve test automatizzato non è sufficiente per determinare la prontezza per l'implementazione. D'altro canto, voi, come sviluppatori, avete bisogno di un meccanismo relativamente rapido e affidabile per sapere che il codice che avete scritto funziona e non interferisce con il resto del sistema. Quindi, come minimo, i vostri test automatizzati dovrebbero dirvi che è molto probabile che il vostro sistema superi il controllo qualità.

Non danneggiare la struttura

Un vero professionista sa che fornire una funzionalità a scapito della struttura è del tutto sbagliato. È la struttura del codice a conferirgli la sua flessibilità. Se compromettete la struttura, compromettete il suo futuro.

Il presupposto fondamentale alla base di tutti i progetti software è che il software debba essere facile da modificare. Se violate questo presupposto creando strutture rigide, indebolite il modello economico su cui si fonda l'intero settore.

In breve: *dovete essere in grado di apportare modifiche senza costi esorbitanti*.

Sfortunatamente, troppi progetti finiscono per impantanarsi nella melma a causa di problemi strutturali. I compiti che prima richiedevano giorni iniziano a richiedere settimane e poi mesi. La dirigenza, disperata per recuperare il tempo perduto, assume altri sviluppatori per velocizzare le cose. Ma questi sviluppatori non fanno altro che aggiungere altro fango, aggravando il danno alla struttura e aumentando la lentezza.

È stato scritto molto sui principi e sui modelli di progettazione software che supportano strutture flessibili e manutenibili ([PPP2001]). I veri professionisti dello sviluppo di software considerano questi aspetti e si sforzano di modellare in modo adeguato il proprio software. Ma c'è un trucco per farlo che pochissimi sviluppatori seguono: *se volete che il vostro software sia flessibile, dovete provare a "fletterlo"*!

L'unico modo per dimostrare che il vostro software è facile da modificare consiste nell'apportarvi modifiche in modo facile. E quando scoprite che le modifiche non sono così facili come pensavate, perfezionate la sua struttura in modo che la modifica successiva sia più facile.

Quando fare queste semplici modifiche? *Sempre*! Ogni volta che osservate un modulo, apportategli piccole modifiche per migliorarne la struttura. Ogni volta che leggete il codice, modificate la struttura.

Questa filosofia è anche chiamata *merciless refactoring*: "refactoring senza pietà". Io la chiamo "regola dei Boy Scout": lasciate sempre un modulo più pulito di come lo avete trovato. Fate sempre un piccolo gesto di gentilezza verso il codice ogni volta che lo prendete in mano.

Questo è del tutto in contrasto con il modo in cui la maggior parte delle persone considera il software. Pensano che apportare una serie continua di modifiche a del software

funzionante sia *pericoloso*. Non è così! Al contrario, è pericoloso consentire al software di rimanere statico. Se non lo si "flette", quando poi *dovrete* modificarlo, lo troverete "inflessibile".

Perché la maggior parte degli sviluppatori evita di apportare continue modifiche al proprio codice? Perché hanno paura di rovinarlo! E perché hanno paura di rovinarlo? Perché non hanno i test.

Tutto, quindi, riconduce ai test. Se avete una suite di test automatizzati che copre praticamente il 100% del codice, e se quella suite di test può essere eseguita rapidamente a piacere, *allora non avrete alcun timore di modificare il codice*. Come potete dimostrare di non aver paura di modificare il codice? Cambiandolo continuamente.

Gli sviluppatori professionali sono così sicuri del loro codice e dei loro test che non si preoccupano per nulla di dovergli apportare modifiche casuali e puntuali. Cambieranno il nome di una classe, per migliorarlo. Noteranno un metodo un po' troppo lungo mentre leggono un modulo e lo suddivideranno, perché è giusto così. Trasformeranno un'istruzione `switch` in un polimorfismo o collasseranno una gerarchia di ereditarietà in una catena di comandi. In breve, trattano il software come uno scultore tratta l'argilla: lo modellano e lo plasmano continuamente.

Etica del lavoro

La vostra carriera è una vostra responsabilità. Non è responsabilità del vostro datore di lavoro assicurarsi che siate appetibili sul mercato. Né è responsabilità del vostro datore di lavoro formarvi o mandarvi a conferenze o comprarvi libri. Queste cose sono una vostra responsabilità. Guai allo sviluppatore che affidi la sua carriera al proprio datore di lavoro. Alcuni datori di lavoro sono disposti ad acquistare libri e a mandarvi a corsi di formazione e conferenze. Bene, vi stanno facendo un favore. Ma non cadete mai nella trappola di pensare che ciò rientri nelle responsabilità del datore di lavoro. Se il vostro datore di lavoro non fa queste cose per voi, dovete trovare un modo per farle voi stessi.

Non è responsabilità del vostro datore di lavoro nemmeno darvi il tempo di cui avete bisogno per imparare. Alcuni datori di lavoro potrebbero concedervi quel tempo. Alcuni datori di lavoro potrebbero perfino pretendere che voi vi prendiate quel tempo. Ma, ancora una volta, vi stanno facendo un favore e dovete essere opportunamente riconoscenti. Tali favori non sono qualcosa che dovreste aspettarvi.

Dovete al vostro datore di lavoro una determinata quantità di tempo e impegno. Partiamo dalle classiche 40 ore a settimana. Queste 40 ore dovrebbero essere spese per i problemi *del vostro datore di lavoro*, non per *i vostri*.

Dovete immaginare di lavorare 60 ore a settimana. 40 per il vostro datore di lavoro e 20 per voi. Nel corso di queste 20 ore dovreste leggere, esercitarvi, imparare e migliorare le vostre competenze.

Mi sembra di sentirvi: "Ma… e la famiglia? E la mia vita? Devo sacrificare tutto per il mio datore di lavoro?".

Non sto parlando di *tutto* il vostro tempo libero. Sto parlando di 20 ore extra a settimana, circa tre ore al giorno. Se usate la vostra pausa pranzo anche per leggere, se ascoltate un podcast nel tragitto casa-lavoro e se dedicate 90 minuti al giorno all'apprendimento di una nuova lingua, il conto torna.

Fate i calcoli. In una settimana ci sono 168 ore. Date al vostro datore di lavoro 40 ore e alla vostra carriera altre 20. Ne restano 108. 56 le lasciamo al sonno, ne restano ancora 52 per tutto il resto.

Forse non volete prendervi questo tipo di impegno.Va bene, ma allora non dovreste considerarvi professionisti. I professionisti dedicano del *tempo* alla cura della loro professione. Forse pensate che il lavoro debba restare al lavoro e che non dovreste portarvelo a casa. Sono d'accordo con voi! In quelle 20 ore non dovete lavorare per il vostro datore di lavoro, ma per voi stessi, per la vostra carriera.

A volte queste due cose sono allineate fra loro. A volte il lavoro che svolgete per il vostro datore di lavoro è molto utile per la vostra carriera. In quel caso, dedicargli alcune di quelle 20 ore può essere ragionevole. Ma ricordate che quelle 20 ore sono per voi. Devono essere utilizzate per migliorarvi come professionisti.

Forse pensate che questa sia la ricetta perfetta per il *burnout*. Al contrario, è una ricetta per *evitare il burnout*. Presumibilmente siete diventati sviluppatori perché siete appassionati di software e il vostro desiderio di essere un professionista è motivato da quella passione. Nel corso di quelle 20 ore dovete fare cose a supporto di quella vostra passione. Quelle 20 ore devono essere divertenti!

Studio del campo

Sapete che cos'è un diagramma Nassi-Shneiderman? Se non lo sapete, come mai? Sapete qual è la differenza fra una macchina a stati di Mealy e una di Moore? Dovreste. Sapreste scrivere un Quicksort senza andare a cercarlo? Sapete che cosa si intende per "analisi della trasformata"? Potreste eseguire una decomposizione funzionale con i diagrammi di flusso dei dati? Che cosa sono i "tramp data"? Avete mai sentito il termine "connascence"? Che cos'è una tabella di Parnas?

Sono moltissime le idee, le discipline, le tecniche, gli strumenti e i termini che costellano gli ultimi cinquant'anni del nostro campo. Quanto ne sapete? Se volete essere professionisti, dovete conoscerne una buona parte ed estendere costantemente le vostre competenze. Perché dovreste sapere queste cose? Dopotutto, il nostro campo non sta progredendo così rapidamente che tutte queste vecchie idee diventano presto irrilevanti? La prima parte di questa domanda sembra ovvia, a prima vista. Di sicuro il nostro campo sta progredendo a un ritmo feroce. È interessante notare, tuttavia, che, per molti aspetti, tale progresso è periferico. È vero che non aspettiamo più 24 ore per una compilazione. È vero che scriviamo sistemi che ormai occupano gigabyte. È vero che lavoriamo incastonati in una rete che si estende su tutto il globo e che fornisce un accesso immediato a tutte le informazioni. Ma è anche vero che usiamo le stesse istruzioni `if` e `while` che impiegavamo cinquant'anni fa. Molte cose sono cambiate. Molte altre no.

La seconda parte della domanda è certamente falsa. Poche idee degli ultimi cinquant'anni sono diventate irrilevanti. Alcune sono state messe da parte, è vero. L'idea dello sviluppo a cascata è certamente caduta in disgrazia. Ma questo non significa che non dovremmo sapere di che cosa si tratta e quali sono i suoi aspetti positivi e negativi.

Nel complesso, tuttavia, la stragrande maggioranza delle idee duramente conquistate negli ultimi cinquant'anni è preziosa oggi come lo era allora. E forse ancora di più ora. Ricordate la maledizione di Santayana: "Coloro che non riescono a ricordare il passato sono condannati a ripeterlo".

Ecco un *elenco minimo* delle cose che ogni professionista del software dovrebbe conoscere.

- Pattern di progettazione. Dovete essere in grado di descrivere tutti i 24 pattern descritti nel libro GOF e avere una conoscenza pratica di molti dei pattern descritti nei libri POSA.
- Principi di progettazione. Dovete conoscere i principi SOLID e avere una buona conoscenza dei principi dei componenti.
- Metodi. Dovete conoscere i metodi XP, Scrum, Lean, Kanban, Waterfall, Structured Analysis e Structured Design.
- Discipline. Dovete praticare lo sviluppo TDD, la progettazione a oggetti, la programmazione strutturata, l'integrazione continua e la programmazione in pairing.
- Artefatti. Dovete sapere come utilizzare UML, DFD, diagrammi di struttura, reti di Petri, diagrammi e tabelle di transizione di stato, diagrammi di flusso e tabelle decisionali.

Apprendimento continuo

Il ritmo frenetico dei cambiamenti nel nostro settore fa sì che gli sviluppatori debbano continuare a imparare nuove cose anche solo per stare al passo. Guai a chi passa all'architettura del software e smette di programmare: ben presto si ritroverà a essere irrilevante. Guai ai programmatori che smettono di imparare nuovi linguaggi: si troveranno sorpassati. Guai agli sviluppatori che non riescono a imparare nuove discipline e tecniche: i loro colleghi emergeranno mentre loro decadranno.
Vi fareste visitare da un medico che non si tiene costantemente aggiornato? Assumereste un avvocato tributario che non si tiene aggiornato con le leggi fiscali e la giurisprudenza recente? Perché i datori di lavoro dovrebbero assumere sviluppatori che non si tengono aggiornati?
Leggete libri, articoli, blog, tweet. Andate alle conferenze. Partecipate ai gruppi di utenti. Entrate nei gruppi di lettura e di studio. Imparate concetti che non rientrano nella vostra zona di comfort. Se siete programmatori .NET, studiate il Java. Se siete programmatori Java, studiate il Ruby. Se siete programmatori C, studiate il Lisp. Se volete davvero rendere flessibile la vostra mente, studiate il Prolog e il Forth!

Esercizi

I professionisti si esercitano. I veri professionisti lavorano sodo per mantenere le proprie competenze ben affilate e pronte. Non basta semplicemente svolgere il proprio lavoro quotidiano e chiamarlo "esercitarsi". Svolgere il proprio lavoro quotidiano è svolgere attività, non esercitarsi. Esercitarsi è quando si mettono alla prova appositamente le proprie competenze *al di fuori* dell'attività lavorativa, con il solo scopo di affinarle e migliorarle. Che cosa potrebbe mai significare, per uno sviluppatore di software, esercitarsi? A prima vista il concetto sembra assurdo. Ma fermatevi un attimo a pensarci. Pensate ai musicisti. Non si esercitano esibendosi. Si esercitano prima di esibirsi. E come si esercitano? Fra le altre cose, svolgono alcuni esercizi particolari. Scale, studi e passaggi. Li ripetono per allenare le dita e la mente e per mantenere la padronanza della loro abilità.
Quindi, che cosa potrebbero fare gli sviluppatori per esercitarsi? In questo libro ho dedicato un intero capitolo alle diverse tecniche di esercitazione, quindi ora non entrerò

troppo nei dettagli. Una tecnica che uso frequentemente è la ripetizione di semplici esercizi come `Bowling Game` o `Prime Factors`. Chiamo questi esercizi *kata*. In questo libro troverete molti kata di questo tipo fra cui scegliere.
Un kata, di solito, si presenta sotto forma di un semplice problema di programmazione da risolvere, come scrivere una funzione che calcoli i fattori primi di un numero intero. Lo scopo del kata non è quello di capire come risolvere il problema; sapete già come farlo. Lo scopo del kata è quello di allenare le dita e la mente.
Io svolgo uno o due kata al giorno, spesso come parte dell'abitudine al lavoro. Potrei farlo in Java, in Ruby o in Clojure o in qualche altro linguaggio per il quale voglio mettere alla prova le mie competenze. Userò il kata per affinare una particolare competenza, come allenare le mie dita a premere tasti di scelta rapida o come l'uso di certi refactoring. Considerate il kata come un esercizio di riscaldamento di 10 minuti al mattino e di defaticamento di 10 minuti alla sera.

Collaborazione

Il secondo modo migliore per imparare è collaborare con altri. Gli sviluppatori professionali cercano appositamente di programmare insieme, di esercitarsi insieme, di progettare e pianificare insieme. In questo modo imparano molto gli uni dagli altri e riescono a fare di più, più velocemente e con meno errori.
Questo non significa che dovete trascorrere il 100% del vostro tempo a lavorare con altri. Anche il tempo di lavoro da soli è molto importante. Per quanto mi piaccia programmare in coppia con altri, ho bisogno di lavorare da solo, di tanto in tanto.

Mentoring

Il modo migliore per imparare è insegnare. Niente vi fisserà i fatti e i concetti in testa più velocemente e solidamente che doverli comunicare alle persone di cui siete responsabili. Quindi il vantaggio dell'insegnamento è fortemente a favore dell'insegnante.
Allo stesso modo, non c'è modo migliore per accompagnare i nuovi arrivati in un'azienda che sedersi con loro e mostrare loro "come funziona". I professionisti si assumono la responsabilità personale di fare da mentore agli junior. Non lasceranno che uno junior operi senza supervisione.

Conoscenza del dominio

È responsabilità di ogni professionista del software comprendere il dominio delle soluzioni che sta programmando. Se state scrivendo un sistema di contabilità, dovete conoscere il dominio della contabilità. Se state scrivendo un'applicazione per viaggi, dovete conoscere il settore dei viaggi. Non dovete diventare esperti del dominio, ma dovete dedicarvi in modo ragionevole al suo studio.
Quando avviate un progetto in un nuovo dominio, leggete uno o due libri sull'argomento. Interrogate i vostri clienti e utenti sui fondamenti e le basi del dominio. Trascorrete del tempo con gli esperti e cercate di scoprire i loro principi e valori.

È fra i peggiori tipi di comportamenti non professionali programmare ciecamente in base a una specifica senza capire il senso di quella specifica. Piuttosto, dovete conoscere abbastanza il dominio per essere in grado di riconoscere e contestare gli errori della specifica.

Identificatevi con il vostro datore di lavoro/cliente

I problemi del vostro datore di lavoro sono i *vostri* problemi. Dovete capire quali sono questi problemi e lavorare per trovare le soluzioni migliori. Quando sviluppate un sistema, dovete mettervi nei panni del vostro datore di lavoro e assicurarvi che le funzionalità che state sviluppando soddisfino davvero le sue esigenze.
È facile per gli sviluppatori "fare gruppo". È facile cadere in un atteggiamento "noi contro loro" rispetto al datore di lavoro. I professionisti evitano in tutti i modi questa situazione.

Umiltà

La programmazione è un atto creativo. Quando scriviamo codice, creiamo qualcosa dal nulla. Stiamo imponendo con coraggio l'ordine al caos. Stiamo controllando con sicurezza, nei minimi dettagli, i comportamenti di una macchina che altrimenti potrebbe causare danni incalcolabili. Quindi, la programmazione è un atto di suprema arroganza.
I professionisti sanno di essere arroganti e rigettano la falsa umiltà. Un professionista conosce il proprio lavoro e ne è orgoglioso. Un professionista ha fiducia nelle proprie capacità e si assume rischi coraggiosi e calcolati basandosi su quella sicurezza. Un professionista non è timido.
Tuttavia, un professionista sa anche che ci saranno momenti in cui fallirà, in cui i suoi calcoli dei rischi saranno sbagliati, in cui le sue capacità saranno insufficienti; e allora si guarderà allo specchio e vedrà il sorriso di un idiota arrogante.
Quando un professionista si ritrova a essere il bersaglio di un brutto scherzo, sarà il primo a riderne. Non ridicolizzerà mai gli altri, ma accetterà il ridicolo quando se lo è meritato e ne riderà quando non è così. Non sminuirà un altro perché ha commesso un errore, perché sa che potrebbe essere proprio lui il prossimo a fallire.
Un professionista accetta la propria suprema arroganza, e che il destino prima o poi se ne accorgerà e livellerà il suo obiettivo. E quando ciò accadrà, la cosa migliore che potete fare è seguire il consiglio di Howard: riderne.

Bibliografia

[PPP2001]: Robert C. Martin, *Principles, Patterns, and Practices of Agile Software Development*, Upper Saddle River, NJ: Prentice Hall, 2002.