

Parte 2: approfondimenti.

In questa sezione si cerca di approfondire alcuni temi di particolare importanza per la corretta applicazione della programmazione estrema.

5. Continua integrazione.

5.1. Strategie di integrazione.

Ogni nuovo pezzo di codice scritto, che sia una modifica del software esistente o una parte scritta ex novo, deve essere integrato nel più breve tempo possibile.

Si è visto nel paragrafo 4.3 che i singoli processi durano al massimo un paio di giorni. La procedura di integrazione deve essere ripetuta più volte durante questo periodo per evitare al massimo i conflitti.

Quando si modifica una parte delle librerie comuni, utilizzata da diversi programmi e sottoprogrammi, è opportuno eseguire subito i test ed integrare, rendendo disponibile a tutti le novità.

In ogni caso l'integrazione non può essere posticipata il giorno successivo. L'ideale sarebbe ripeterla ogni volta che trascorrono alcune ore.

Perché ciò sia possibile devono essere verificate alcune circostanze.

- La programmazione deve essere fatta per piccoli passi, in modo da essere sempre a poche mosse da una versione stabile del software. Anche grosse modifiche devono essere fatte un pezzetto alla volta passando per configurazioni intermedie, saggiando le modifiche una ad una ed evitando di rincorrere errori introdotti chissà quando.
- Gli strumenti per integrare devono essere sufficientemente potenti da permettere una veloce esecuzione del ciclo che comprende integrazione, compilazione, esecuzione dei test. È buona norma riservare la macchina più potente proprio per questo compito ed è

utile disporla in un posto facilmente accessibile da tutti. Magari al centro dell'ufficio.

- Deve esistere un insieme abbastanza completo di test che possa essere eseguito senza troppo impegno da parte del programmatore. L'ideale sarebbe spingere un solo bottone ed aspettare il messaggio finale che deve essere **si** o **no**. Nell'ambiente legato a XP in genere si parla di spia verde (successo) o rossa (fallimento).

Le probabilità di collisioni sono basse se l'integrazione è ripetuta spesso. Se poi si trova un conflitto ed uno o più test fallisce, è abbastanza facile porvi rimedio, poiché le modifiche introdotte sono poche e facilmente circoscrivibili.

È proprio la frequente integrazione ed il modo di lavorare per piccoli episodi che assicura di avere il sistema diviso in tanti piccoli pezzi indipendenti. Questo aiuta anche a programmare correttamente ad oggetti, favorendo l'incapsulamento del codice in piccoli e dunque semplici oggetti.

Ogni metodo è un punto in cui si possono introdurre modifiche senza coinvolgere parti più estese del programma.

Il rischio di conflitti decresce ulteriormente se il gruppo è ben affiatato e ci si attiene a regole di codifica standard. Nell'esperienza che ho avuto durante l'applicazione della programmazione estrema è capitato che due coppie, lavorando su computer diversi, avessero aggiunto un metodo con lo stesso nome. La cosa stupefacente era che anche l'implementazione del metodo era identica e dunque il conflitto semplicemente non esisteva.

Un altro importante aspetto positivo della continua integrazione è che se due persone la pensano diversamente su come risolvere un dato problema, questa divergenza viene subito a galla.

Infine si deve sottolineare che vengono praticamente azzerati i rischi di avere un sistema instabile nel momento in cui i manager decidano che sia il

momento di cambiare, oppure l'utente abbia bisogno di lavorare sul prodotto anche se incompleto. Il sistema contiene in ogni momento un sottoinsieme delle funzionalità richieste già completamente funzionanti e tale sottoinsieme comprende quelle di maggiore importanza e di maggior valore.

5.2. Uno strumento per integrare. CVS.

Quando si programma bisogna credere di essere gli unici a stare lavorando sul progetto. Il sistema su cui siamo non deve in nessun modo cambiare senza il nostro intervento. Per questo motivo è utile lavorare su una copia dell'intero progetto.

Anche se qualche altro sta integrando nuove modifiche, la nostra copia rimane intatta e gli effetti prodotti dal programma dipendono solo dalle nostre modifiche, dunque sono facili da individuare.

Lavorare in questo modo ci permette anche di fare modifiche su parti del sistema più o meno distribuite senza influenzare direttamente il lavoro di altri.

Per poter lavorare in questo modo è necessario un potente strumento per archiviare le varie versioni dei file, per confrontarle, ripristinarle in caso non passino i test.

Per fare tutto ciò noi ci affidavamo **CVS** (Concurrent Versions System).

CVS nasce per andare incontro alle esigenze dei primi sviluppatori open-source. Rendere pubblico un proprio programma significa prendersi la responsabilità di mantenerlo, correggere i bachi, aggiungere aggiornamenti, insomma rispondere continuamente alle richieste che arrivano e che possono essere anche numerose.

Il primo strumento utilizzato da questi sviluppatori è stato il comando UNIX **diff**, che permette di confrontare due file e trovare le differenze. Un miglioramento di questo strumento si ha con **patch**: un comando che prendendo l'output di diff ed uno dei due file riesce a ricostruirsi il secondo.

Questi due strumenti sono molto comodi per riuscire a risalire a diverse versioni di un file memorizzando unicamente le differenze.

Se le versioni iniziano ad essere molte, anche questi due strumenti sono insufficienti. È necessario mantenere una storia della sequenze di modifiche per riuscire così ad eliminarne una se ci si accorge che questa ha introdotto bachi che si sono rivelati solo dopo l'introduzione di tante altre modifiche.

Una prima valida soluzione è stato il tool RCS (Revision Control System), utilizzato anche da aziende private. Uno dei suoi problemi è che era centrato attorno ai singoli file. Non si aveva il concetto di progetto. Un'altra limitazione derivava dalla gestione dei lock sui file utilizzati al momento.

Una sola persona alla volta poteva accedere e modificare un file e spesso ci si scordava di rilasciare i lock con il risultato di avere sviluppatori in perpetua attesa.

Per finire, tutto ciò doveva essere fatto su una unica macchina perché RCS non era distribuito.

Per risolvere questi problemi e la continua necessità di negoziazione tra gli sviluppatori nel 1986 venne inventato CVS, che non è altro che una evoluzione di RCS.

Dal 1990 CVS può anche essere applicato ad un ambiente distribuito. Per questo motivo viene oggi ampiamente utilizzato dalle comunità open-source che lavorano in internet ed anche da innumerevoli ditte private all'interno delle reti aziendali.

L'idea che sta dietro a CVS parte dal concetto che una release è solo uno snapshot dell'intero progetto, relativo ad una certa data. Un progetto open-

source è in un certo modo in continua fase di release, dato che ognuno può aggiungere modifiche.

I metodi tradizionali per la gestione del software erano stati disegnati con l'idea che la release è un evento monumentale che chiude un lungo ciclo di lavoro.

Questo sicuramente non può essere applicato ad un progetto open-source, né tanto meno ad un progetto che si svolge secondo le regole di XP: il tempo che passa tra una release e l'altra deve essere il più breve possibile.

Noi non utilizzavamo direttamente CVS, ma tramite una comoda e trasparente interfaccia grafica: CVL (Concurrent Versions Librarian) sviluppato da Sente.

Tramite essa è possibile gestire in modo quasi automatico tutta la fase di integrazione. Vengono segnalati tutti i file modificati in locale rispetto al computer su cui vengono tracciate tutte le varie versioni. Viene fatto il merge in automatico, segnalando eventuali conflitti se più persone hanno modificato contemporaneamente le stesse linee di codice. In tali casi si possono confrontare i vari file evidenziando le differenze.

Di ogni modifica viene registrata l'ora e l'autore.

Prima di iniziare a modificare file sulla copia locale questa può facilmente essere aggiornata con l'ultima versione presente sul server, oppure con una qualsiasi precedente.

Di uno stesso progetto possono esserci più versioni attuali e si può lavorare indipendentemente su ciascuna: si possono costruire rami storici diversi a partire da un unico progetto. Questo può essere utile se per esempio esistono più utenti con esigenze differenti.

L'uso di CVS semplifica anche la procedura di back-up dei sorgenti. È sufficiente conservare l'unica cosa importante che contiene tutti gli aggiornamenti fino al massimo a qualche ora fa, in pratica il disco su cui gira CVS.

Alla sera possiamo andare a casa tranquilli senza avere paura di perdere il nostro lavoro, inoltre grazie alla continua integrazione, la nostra mente non contiene lavori lasciati a metà: tutto quello che è stato implementato ed integrato deve per forza funzionare. Vista la brevità degli episodi, raramente rimangono modifiche sospese.

6. Proprietà collettiva.

Abbiamo appena visto che CVS è uno strumento essenziale anche per applicare in modo proficuo la proprietà collettiva del codice.

Avere codice condiviso assicura di non dover aspettare, che qualcuno applichi delle modifiche a codice di *sua proprietà* per iniziare ad implementare le proprie idee.

Per praticare la proprietà collettiva con successo è assolutamente necessario avere test per ogni pezzo di codice, altrimenti lasciare ad ognuno la possibilità di modificare l'intero sistema porterebbe solo al caos.

Abbiamo anche visto che la continua integrazione è un fattore importante per rendere le nuove modifiche disponibili a tutti i programmatori, e permette di rilevare tempestivamente i conflitti.

Un effetto positivo della proprietà collettiva è che le parti di codice complesso non sopravvivono a lungo.

Quando si continua a lavorare per più ore sullo stesso codice, non si riesce a comprendere bene se le modifiche che continuiamo ad introdurre siano leggibili o meno. Ormai conosciamo troppo bene questa parte di codice ed a noi sembra sempre tutto chiaro.

Appena un'altra coppia di programmatori ha sott'occhio lo stesso codice si accorge subito se alcune parti sono più complesse e più difficili da leggere. Immediatamente la complessità è rimossa senza compromettere il funzionamento grazie all'ausilio dei test.

Tutto questo può essere visto anche in un altro verso: poiché sappiamo che il nostro codice verrà in breve letto da tutto il gruppo, sicuramente quando scriviamo stiamo molto attenti a non introdurre complessità che difficilmente possa essere giustificata.

Rendere tutto il codice pubblico assicura che non esista nessuna parte del codice conosciuta solo da uno o due programmatori. Questo riduce enormemente i rischi del progetto. Infatti, anche se uno dei programmatori si ammalasse, non ci sarebbero gravi problemi. La possibilità che su un solo programmatore ricada tutto il lavoro attuale è addirittura un evento che non è nemmeno pensabile.

Il risultato è avere incrementato la comunicazione all'interno del gruppo in modo implicito: la conoscenza del sistema si diffonde molto velocemente.

7. Programmazione in coppia.

7.1. Motivazioni.

Programmare in coppia è una pratica che porta innumerevoli vantaggi, anche se attuata al di fuori di un gruppo che segue le regole di XP.

Aderire a XP non è essenziale per lavorare in coppia, eppure attualmente il maggior contributo al numero di persone che applicano programmazione in coppia è dato proprio dalla programmazione estrema.

La maggior obiezione fatta alla programmazione in coppia è lo spreco di risorse umane necessario per implementarla.

Numerosi esperimenti hanno provato che ciò non è vero. Programmando in coppia si riesce ad incrementare notevolmente l'efficienza del processo di produzione e la qualità del software prodotto, a scapito di un piccolo overhead.

Facendo considerazioni a lungo termine anche questo piccolo overhead sparisce ed anzi si guadagna tempo.

Programmare in coppia ha un grande effetto positivo per la comunicazione e l'importanza di questa è sottolineata in XP.

Il fatto che le coppie siano spesso mischiate aumenta ancora il diffondersi della conoscenza all'interno del gruppo.

Tra i due programmatori che formano una coppia, uno controlla la tastiera ed il mouse ed è a lui assegnato il compito di scrivere il codice. L'altro continuamente ed attivamente osserva, cerca difetti, considera quali alternative esistano, controlla l'utilizzo delle risorse, si occupa delle implicazioni strategiche.

Contestualmente si instaura un continuo dialogo ed un continuo scambio di idee. Periodicamente e regolarmente i due invertono i ruoli.

Lavorare assieme è anche il miglior modo per imparare e crescere. Programmatori poco esperti possono in breve tempo diventare validi compagni anche per i programmatori più abili e più smaliziati.

Per un programmatore alle prime armi è sicuramente il modo migliore per imparare velocemente tecniche che altrimenti richiederebbero anni di esperienze. Anche il programmatore esperto può trarne parecchi vantaggi: evita di fossilizzare il proprio modo di programmare ed apre la mente a nuove idee ed a nuovi modi di pensare.

In fondo più tempo si ha alle spalle come programmatori, più si è consapevoli che quello che conosciamo è un'infima parte di quello che potrebbe servirci conoscere. La tecnologia si evolve talmente velocemente che è arduo, se non impossibile, essere sempre aggiornati sugli argomenti che stiamo trattando.

Una coppia riesce a trovare molte più soluzioni ai problemi che il singolo e dunque più facilmente viene trovata la soluzione più efficiente e qualitativamente migliore.

Se è vero che XP non è essenziale per applicare correttamente la programmazione in coppia, viceversa senza quest'ultima XP non potrebbe risultare vincente.

Sotto stress, le persone regrediscono. Vengono abbandonate progressivamente le pratiche. La scrittura dei test viene posticipata alla scrittura del codice, per poi essere saltata del tutto. La ristrutturazione è eseguita sempre più di rado e mai quando se ne sente più il bisogno, perché proprio in quei momenti lo stress e la fretta sono maggiori.

L'integrazione viene ritardata per "risparmiare tempo" senza accorgersi che poi integrare diventerà più arduo e la velocità dell'intero gruppo calerà.

La presenza di un compagno ci aiuta nei momenti difficili. Se si è in due è più facile che uno dei due mantenga la calma necessaria ed assieme è più probabile seguire le buone regole. In due si può combattere la paura ed acquisire fiducia in ciò che ci si appresta a fare.

Un'ultima fondamentale qualità della programmazione in coppia è che rende il lavoro più divertente ed aumenta la soddisfazione personale.

Vengono soddisfatti i bisogni di avere contatti umani, evitando di isolarsi a lavorare sempre e solo con il computer.

Il livello di concentrazione e d'attenzione rimane costantemente alto e si perde meno tempo in attività inutili.

Mantenere la concentrazione ad alto livello per più di otto ore al giorno può essere controproducente. Per questo motivo sono necessarie pause in cui ci si distrae e ci si rilassa, inoltre è utile anche lavorare regolarmente un massimo di quaranta ore settimanali.

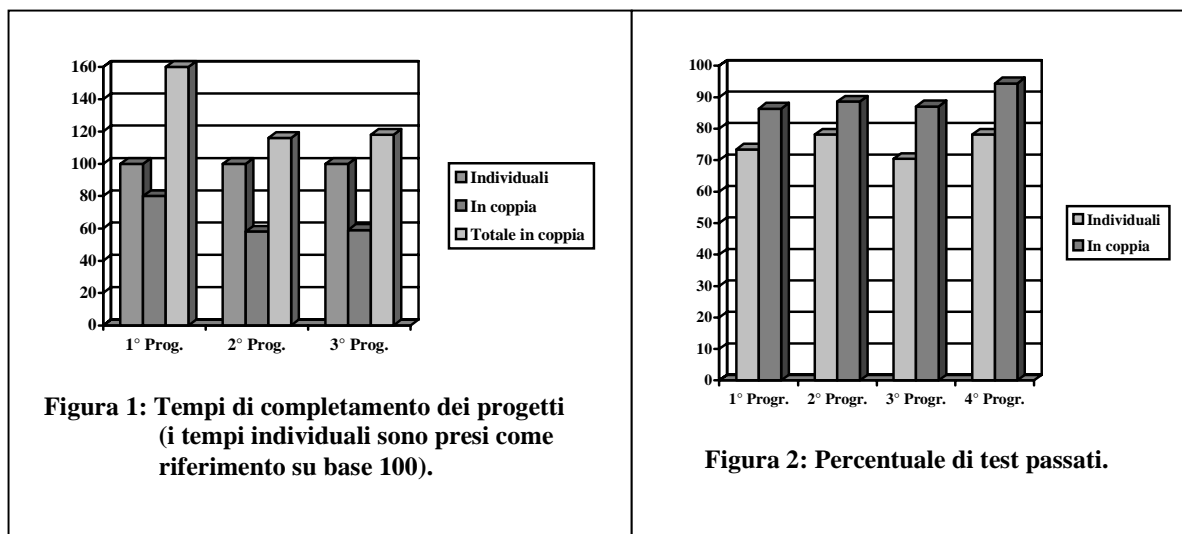
7.2. Esperimenti sulla programmazione in coppia.

Nel 1998 un professore della Temple University ha svolto un esperimento su 10 coppie di programmatori professionisti confrontati con 5 programmatori singoli.

In un altro esperimento del 1999 14 coppie di studenti dell'University of Utah vennero confrontate con altri 13 studenti singoli. Si fece in modo che i due gruppi di studenti fossero equamente assortiti.

Le misure vennero fatte tramite strumenti imparziali: si misurava il tempo impiegato per risolvere i problemi sottoposti e si faceva un controllo sulla qualità tramite test automatici.

I due esperimenti hanno mostrato risultati simili.



I tempi di produzione delle coppie è risultato subito inferiore a quello dei singoli, anche se la somma totale del tempo speso dai singoli era minore.

Nei successivi esperimenti le coppie, aumentando il livello di coordinazione, riuscivano ad avvicinare di molto i tempi totali dei singoli.

La qualità del software prodotto era nettamente più alta per le coppie e migliorava al crescere dell'affiatamento dei due componenti.

Se si considera che, nel mercato d'oggi, avere prodotti di qualità il più velocemente possibili, anche a costi leggermente più alti, può essere la chiave che distingue il successo dall'insuccesso, allora è evidente come la strategia di lavorare in coppia possa diventare vincente.

Anche nella programmazione in coppia possono sorgere delle difficoltà. Se uno dei compagni ha una personalità troppo forte, può rischiare di prevalere e di imporre le sue idee, soffocando l'altro.

Viceversa un compagno troppo timido non contribuisce sufficientemente al risultato finale.

Anche in questi casi la programmazione in coppia può risultare utile per insegnare come collaborare proficuamente, aiutando gli uni a contenersi e gli altri a tirar fuori le proprie idee ed a partecipare attivamente.

Per questo tipo di problemi è utile ruotare spesso le coppie.

8. Ristrutturazione.

8.1. Tanti piccoli passi.

La ristrutturazione è il processo che permette di fare modifiche ad un sistema software in modo che non sia alterato il comportamento esteriore, ma sia perfezionata la sua struttura interna.

È un modo disciplinato per *pulire* il codice che minimizza le probabilità di introdurre bug in programma già funzionante.

Le metodologie classiche insegnano che prima si disegna il sistema, poi si inizia a codificare. Ristrutturare il programma significa invece aggiungere una parte del disegno per volta. Con il passare del tempo il codice viene modificato ed il disegno originario non è più rispettato. Per questo bisogna rivedere il disegno ed adattarlo passo per passo alle nuove funzionalità.

Ogni mossa che si esegue è semplice, se non addirittura banale, per evitare di modificare involontariamente il comportamento del programma.

Si sposta una variabile da una class ad un'altra, si estrae una parte di codice da un metodo per crearne un altro, si sposta un po' di codice verso i livelli alti di una gerarchia per astrarre un concetto, si divide una classe in due, si elimina un commento spostando la parte di codice in un metodo con un nome significativo, si rinomina una variabile o un metodo per rendere meglio il suo scopo. Eppure l'effetto totale di queste piccole modifiche può migliorare radicalmente il disegno, anche partendo da una situazione di completo caos.

8.2. Refactoring e programmazione ad oggetti.

Quando ho conosciuto per la prima volta la programmazione ad oggetti ho subito pensato a quanto fosse difficile creare una gerarchia di oggetti.

Osservando gli esempi fatti dal professore mi spaventava l'idea che a me non sarebbero mai venute in mente soluzioni del genere. In realtà programmare ad oggetti non implica conoscere subito la soluzione, ma arrivarci progressivamente.

Ad ogni piccolo passo i dati e le funzioni si aggregano lentamente attorno a determinati centri gravitazionali fino a dare vita a nuovi oggetti. Un oggetto può disgregarsi e dare vita a più oggetti.

I pionieri della programmazione OO erano ignari di ciò che avrebbero potuto ottenere. Oggi invece sono stati classificati numerosi problemi per cui c'è già una soluzione pronta. Alcuni programmatori che si sono accorti di risolvere ripetutamente lo stesso tipo di problema, hanno iniziato a catalogare i **pattern**.

Ad esempio nell'esame di *Linguaggi e Traduttori* ho ampiamente utilizzato il *pattern visitor*, utilissimo per implementare costrutti come analizzatori lessicali, sintattici e semantici partendo da una gerarchia comune.

Quasi tutti durante i primi approcci alla programmazione ad oggetti in genere continuano a programmare come se gli oggetti fossero delle grosse scatole e non degli utili strumenti di sviluppo per risolvere dei problemi e realizzare un codice di più alta qualità.

Imparare a conoscere uno alla volta strumenti come la località, l'eredità, la delegazione,... permette di scoprire a piccoli passi tutte le potenzialità insite nella programmazione ad oggetti.

La ristrutturazione ci permette di applicare i vari pattern ad un programma scritto in un cattivo codice ad oggetti e trasformarlo in un programma che sfrutta tutte le risorse della programmazione ad oggetti.

È impossibile costruire un framework perfetto al primo tentativo. Con la ristrutturazione si possono organizzare le classi in coerenti gerarchie di classi.

Un'ultima fatto che voglio sottolineare è che programmazione ad oggetti e refactoring si sostengono a vicenda.

8.3. Occasioni per ristrutturare.

Quando dobbiamo aggiungere una nuova funzionalità al programma ed il codice non è strutturato in modo tale da favorire l'introduzione, allora è necessario prima fare un po' di ristrutturazione.

Durante il quotidiano lavoro di programmazione ci possiamo accorgere che è necessario ristrutturare osservando chiari segni che il codice evidenzia.

Metodi troppo lunghi possono essere suddivisi in metodi più piccoli e più significativi, oppure possono dare vita ad un nuovo oggetto.

Metodi con troppi parametri sono poco leggibili. Si può ovviare trasformando una variabile locale in una variabile d'istanza, sostituire un parametro con un metodo, introdurre un oggetto che contenga tutti i parametri affini.

Bisogna assolutamente evitare di avere codice duplicato. È un chiaro segnale che è necessaria una ulteriore astrazione per unificare parti diverse del programma. Sarà così più semplice e veloce modificare quell'unica parte di codice che rimane.

Classi troppo estese possono essere spezzate in più classi, oppure possono essere sviluppate in sottoclassi, ognuna responsabile di una astrazione.

Lunghi statement switch possono essere rimpiazzati tramite polimorfismo o strutture particolari quali ascoltatori, visitor.

La maggior parte di queste modifiche hanno un fisso elenco di regole e di passi che devono essere seguiti per poterle applicare. Alcuni sviluppatori hanno catalogato i casi in cui si può applicare il refactoring ed hanno esplicitato il modo per eseguirlo.

Partendo da questi cataloghi si stanno sviluppando tool per eseguire il refactoring in automatico.

Questi strumenti rendono il lavoro di ristrutturazione più agevole e veloce, minimizzando la probabilità di errore e presentano utili funzionalità tipo *l'undo*.

Sono ancora poco diffusi, ma la tendenza è integrarli nell'ambiente di programmazione, assieme ad altre funzioni automatiche già presenti ed ampiamente utilizzate come la ricerca di riferimenti ad una certa variabile o ad un certo metodo e la sostituzione automatica.

La tendenza è fare in modo che le ricerche possano essere fatte anche a livello semantico e non solo sintattico: un metodo ed una variabile con lo stesso nome sono due entità diverse, così come il nome di una classe o di una variabile locale possono essere ripetuti senza conflitti.

8.4. Refactoring e test.

Ristrutturare può essere molto pericoloso. Partiamo da un programma che funziona e vogliamo ottenere un codice più leggibile, ma nessuno ci assicura che alla fine il programma funzioni ancora.

Anche se i passi sono semplici, per evitare di introdurre bug è necessario avere un insieme di test automatici prima di iniziare a ristrutturare.

Fare i test significa automatizzare il controllo degli errori. È un po' come avere un compilatore configurabile e programmabile a cui possiamo dire cosa controllare.

Si può affermare che i test introducano un nuovo stadio di controllo per soddisfare le nuove esigenze di elevare la qualità sia del software prodotto che della sua produzione.

La ristrutturazione non deve essere fatta a tutti i costi. Possiamo ristrutturare finché il sistema resta sotto il nostro controllo e siamo sicuri di riuscire ad organizzare meglio il nostro codice. Appena ci sorgono dei dubbi e non abbiamo una chiara visione dell'organizzazione del sistema dobbiamo fermarci ed accontentarci dei risultati ottenuti.

Se siamo sicuri che il codice può essere ancora migliorato, ma non sappiamo bene come, possiamo solo rimandare la ristrutturazione. Magari la mattina dopo possiamo avere le idee più chiare per comprendere quello che il giorno prima sembrava impossibile da domare.

Oppure possiamo lasciare che la prossima coppia che dovrà occuparsi di questa parte del codice riesca a comprendere meglio quello che a noi attualmente sfugge. Introducendo una nuova funzionalità potrebbe diventare più chiaro il disegno che ci serve.

Mentre si ristruttura è facile lasciarsi trasportare dall'entusiasmo e procedere attraverso diverse modifiche senza integrare e controllare che i test funzionino. Se poi alcuni test falliscono, correggerli può portare via preziose ore per il debugging.

L'unico modo è fare una modifica alla volta e quindi integrare. Meglio gettare via del codice che ora non funziona e ripartire da una configurazione stabile, che perdere parecchie ore e soprattutto la pazienza.

Riscrivere da capo un pezzo di codice non porta mai via troppo tempo. Sicuramente la fatica è molto minore di quella fatta per setacciare il programma e trovare il baco introdotto: spesso una semplice svista.

Quando si ristruttura non bisogna preoccuparsi delle prestazioni del sistema. Lo scopo del refactoring è rendere il codice più leggibile e modificabile. A volte si riesce a rimuovere la metà del codice grazie alle semplificazioni.

Una volta che si è finito di ristrutturare, diventa più facile adeguare il sistema alle esigenze di prestazioni ed effettivamente ottimizzarlo.

Ristrutturare può essere anche un modo per scoprire errori all'interno del codice nascosti in strutture troppo complesse facendoli emergere.

È infine un modo per chiarire le nostre idee e le assunzioni che abbiamo fatto sul codice.