

# Introduzione.

Il fatto che ogni anno l'hardware sul mercato sia più veloce e più sofisticato di quello dell'anno precedente è ben noto (legge di Moore).

La possibilità di integrare un numero maggiore di transistori su un unico chip permette di avere componenti sempre più complessi.

Questa complessità deve essere però gestita correttamente ed implica uno sforzo notevole per trasferirla sulle prestazioni finali. Un progetto non adatto ad una complessità maggiore può portare ad un componente con conflitti interni, funzioni replicate inutilmente e slegate tra loro, anziché ad un componente più veloce e con più funzionalità.

Altro fatto ben noto è che il continuo sviluppo dell'hardware permette al software di evolversi in maniera analoga.

Se abbiamo un programma che sulle macchine attuali gira senza problemi, fra sei mesi potremmo già vedere delle degradazioni nella velocità installando la versione successiva.

Questa rincorsa fra hardware e software porta ad avere sistemi sempre più complessi che devono poter essere modificati nel più breve tempo possibile. Oggi può servire ottimizzare agli estremi una particolare funzione per rendere il programma veloce, mentre fra un anno questa ottimizzazione risulta inutile perché l'hardware riesce a sopperire alla differenza di velocità.

Gli sforzi si possono così spostare verso migliore funzionalità, modularità, leggibilità, a scapito dell'efficienza non più necessaria; caratteristiche nuove che permettono di sviluppare software in tempi minori elevando il livello della qualità.

Si è parlato di velocità, ma lo stesso vale per tutte le altre caratteristiche di un computer, soprattutto per quanto riguarda le dimensioni della memoria: il software ha a disposizione una quota sempre maggiore di RAM per poter essere eseguito ed un maggiore spazio su hard-disk per l'installazione.

La maggior parte del software esistente è stato prodotto prendendo decisioni a breve termine senza seguire un progetto di sviluppo.

Questa strategia va bene per piccoli programmi, ma quando un programma cresce e si evolve, allora aggiungere nuove funzionalità diventa sempre più arduo, il sistema diventa intricato. È facile aggiungere un numero crescente di errori e la loro rilevazione diventa più difficile.

La continua crescita delle **dimensioni del software** e della **velocità con cui si modificano gli obiettivi** suggerisce che il processo di produzione debba essere curato attentamente. Questo ha portato alla nascita di **metodologie di programmazione** diverse, nate per rispondere ad esigenze diverse.

Le **metodologie classiche** prevedono un rigido insieme di regole da seguire ed una rigida organizzazione per ovviare a questi problemi.

Durante lo sviluppo un sistema parte con una struttura relativamente semplice: all'inizio esiste solo lo schema mentale in testa ai progettisti e i primi diagrammi di analisi; poi il passaggio dall'analisi al progetto ed alla successiva codifica apporta nuovi elementi di complessità; inoltre lo sviluppo avviene di solito in modo incrementale, aggiungendo nuovi moduli a quelli del nucleo iniziale.

Una volta rilasciato, il sistema evolve nel tempo e necessita di manutenzione. Ciò comporta un ulteriore incremento della sua complessità, in termini di aggiunte di moduli o di modifiche alle funzionalità esistenti, con riscritture parziali del codice e pericolo di effetti collaterali su altre parti. In tutti i casi, l'*entropia* del sistema aumenta col tempo, così come il costo di ogni nuovo sforzo di modifica. L'ingegneria del Software ha dimostrato che tale crescita è esponenziale. È questo un altro modo per presentare ciò che sin dagli anni '60 è chiamata la **crisi del Software**.

Da qualche anno si stanno affacciando **nuove metodologie**, cosiddette *leggere* contrapposte a quelle classiche, spesso riferite con l'appellativo di *pesanti*. A queste nuove metodologie leggere appartiene **Extreme Programming**, che sarà al centro della discussione di questa tesi.

# 1. Una panoramica sulle metodologie.

## 1.1. Processi pesanti e metodologie predittive.

Tra le metodologie più note ed utilizzate ci sono quelle che impongono un processo disciplinato di sviluppo del software con lo scopo di renderlo **predicibile** ed **efficiente**.

Questo scopo è stato raggiunto dalle *metodologie* cosiddette *pesanti* introducendo processi molto dettagliati e pragmatici. Si riesce così a pianificare l'intero progetto, a dividerlo in singoli processi, ognuno dei quali con una scadenza determinata ed un budget fissato.

Viene inoltre stabilito un insieme di regole che i programmatori devono rigidamente seguire. Questo tende a limitare le capacità dei singoli ma porta tutte le persone verso uno standard richiesto.

Macroscopicamente il progetto è diviso in due differenti attività:

- la **pianificazione** che deve essere gestita da persone creative e capaci di risolvere problemi complessi,
- la fase di **costruzione** che segue la pianificazione e diventa relativamente facile e predicibile.

Si tenta di ridurre la necessità di modifiche al sistema tramite un'analisi dei requisiti molto accurata, magari condotta con l'approccio dei casi d'uso, e di facilitare la comprensione del sistema, e quindi le sue modifiche, tramite la stesura della documentazione a livello di analisi e progetto fatta con una notazione standard (**UML**) e con strumenti automatici.

Spesso però non risulta facile prevedere come sarà il software finale mentre si pianifica, infatti non è facile prevedere il funzionamento e le funzionalità di un programma senza aver scritto una linea di codice. Inoltre

può non risultar diretta la conversione da UML a software.

Purtroppo tale approccio funziona bene se i requisiti del sistema sono relativamente stabili e se si ha il tempo e la forza economica per imporre l'uso di un processo ben definito, con molti documenti da mantenere allineati al codice e con costosi strumenti automatici. Oggi le applicazioni devono essere sviluppate alla velocità di **Internet**, con requisiti in continuo cambiamento e con Budget spesso limitati, soprattutto per aziende emergenti ancora relativamente giovani e con poca esperienza.

Una cosa molto difficile è riuscire a fare una stima corretta del costo e del ritorno monetario che può avere una certa funzionalità finché non viene effettivamente realizzata ed utilizzata. Conseguenza di questo è una continua necessità di modificare le specifiche.

La tendenza delle metodologie pesanti è quello di sfavorire i cambiamenti e ad avere una notevole inerzia. Una volta che si è finito di pianificare bisogna solo costruire ed ogni modifica comporta un incremento nella spesa che è proporzionale alla fase di avanzamento del progetto e dunque è da evitare il più possibile.

Sicuramente i successi di questa logica sono evidenti rispetto al caotico procedere dei programmatori che non utilizzano nessuna metodologia e risolvono solo i problemi a breve termine. Il costo di questa metodologia non è però da sottovalutare.

Ci sono fattori che comportano un rallentamento notevole nello sviluppo come la necessità di documentare ed approvare ogni minima modifica.

La maggior critica ricevuta riguarda la troppa burocrazia necessaria, l'obbligo di riempire pagine e pagine di documentazione che deve rispecchiare determinati standard stilistici e che molto probabilmente nessuno andrà mai a leggere a causa del volume spropositato di informazioni.

Un loro limite è che tendono a pianificare una grande quantità di software per lunghi lassi di tempo. Questa è una buona cosa finché gli obiettivi, le necessità e le specifiche non cambiano frequentemente.

Una metodologia non può essere considerata valida in assoluto, ma bisogna considerare le "condizioni al contorno" in cui questa viene applicata. Le metodologie pesanti richiedono che le specifiche siano predicibili e stabili, altrimenti la pianificazione tenderà ad allontanarsi sempre più dalla realtà e dalle effettive necessità.

Un altro aspetto negativo delle metodologie pesanti è lo scarso peso dato all'abilità dei programmatori di inventare soluzioni. Il loro lavoro deve essere un lavoro di routine e dunque noioso. Si perdono così degli enormi potenziali di valore.

## 1.2. Processi leggeri e metodologie adattative.

Come reazione a questo modo di programmare negli ultimi anni è apparso un nuovo gruppo di metodologie etichettate come **leggere**.

La differenza più immediata è che queste ultime sono meno orientate alla documentazione e mettono invece il codice al centro della programmazione.

La differenza profonda è invece che queste metodologie sono **adattative**: i cambiamenti sono ben accetti ed anzi la pianificazione viene creata man mano che il progetto cresce e si evolve.

Pianificazione e costruzione vengono a fondersi, supportandosi a vicenda.

Questo risponde alla necessità di avere un continuo feedback che si realizza tramite *iterazioni di sviluppo*, continuo ritorno sui passi e sulle scelte fatte: ad una piccola fase di pianificazione segue una fase in cui si costruisce un prototipo o una prima versione funzionante. Grazie a questa

si può ritornare ad una nuova breve fase di pianificazione che permetta di correggere le funzionalità esistenti e di aggiungerne altre.

Più è breve la durata dell'iterazione, più il **feedback** è forte.

Le metodologie leggere cercano di lavorare assieme e non contro la natura dell'uomo: vengono riconosciuti ed arginati i limiti umani e nello stesso tempo vengono esaltate le caratteristiche positive, l'inventiva.

Si mette l'uomo al centro della programmazione e si considerano anche che la natura umana non è essa stessa predicibile: numerosi fattori possono influenzare la produttività e vanno analizzati.

Si tende a dare più fiducia ai programmatori e più potere decisionale, assegnargli più responsabilità. Un programmatore deve poter prendere decisioni tecniche in base alle informazioni che gli arrivano da uno stretto contatto con il personale manageriale.

Manager e programmatori sono esperti in campi diversi che devono collaborare e mettere in campo le proprie abilità per portare al successo il progetto. Ognuno deve potersi fidare delle decisioni che l'altro prende.

Abbiamo detto che per sapere se una metodologia sia vincente bisogna considerare anche le condizioni al contorno. Un processo adattativo è indicato quando si hanno:

- Specifiche incerte e volatili
- Sviluppatori responsabili e motivati
- Utente flessibile, che riesca ad essere coinvolto ed a capire

Vedremo che questi sono i punti da cui parte Extreme programming.

## 2. Extreme programming.

### 2.1. Cosa significa Programmazione Estrema?

Extreme programming, o più semplicemente **XP**, nasce in risposta alle nuove esigenze di chi produce software, alla cultura del *just-in-time* che ha fortemente preso piede, ai cicli di realizzazione sempre più compressi, all'alto rischio tecnico insito in ogni nuovo progetto.

In questa situazione il cambiamento e l'adattamento devono diventare un'abitudine.

XP non presenta nulla di sconvolgentemente nuovo, anzi la maggior parte delle tecniche sono già state inventate, sperimentate e magari abbandonate. XP è innovativa perché cerca di metterle assieme in un modo che assicuri la loro migliore applicazione.

Ogni tecnica deve essere sostenuta e controllata dalla contemporanea applicazione delle altre. Si riesce così a sopperire ai limiti di ognuna di queste presa singolarmente.

XP è adatta per gruppi medio-piccoli di programmatori che sviluppino software con specifiche in continua e rapida evoluzione.

Secondo Kent Beck il gruppo deve essere composto da 2-10 persone; per altri (es: Martin Fowler) si può arrivare anche a 40.

È necessario che l'ambiente di programmazione non sia troppo vincolante e che permetta inoltre di eseguire i test in un tempo non eccessivamente lungo.

Il nome Extreme Programming nasce dal tentativo di portare all'estremo le proprietà e l'applicazione del senso comune e della semplicità.

XP si basa su alcuni semplici principi:

- Rivisitazione e ristrutturazione del codice (refactoring)
- Costruzione di gruppi di test sia legati al codice che funzionali
- Scelta della cosa più semplice che lavori
- Esecuzione dei test ed integrazione svolti più volte al giorno, in modo da avere un forte feedback
- Iterazioni brevi (ore, al massimo giorni).

XP si pone lo scopo di ridurre il rischio insito nella produzione del software, di permettere un continuo e rapido cambiamento degli obiettivi del progetto, di aumentare la produttività . . . e di portare maggior divertimento e soddisfazione tra i programmatori.

È centrata sull'uomo. Cerca di limitare le sue debolezze e di far emergere le sue capacità. Per questo si preoccupa anche della soddisfazione e realizzazione del programmatore.

La fretta e la stanchezza inducono facilmente a commettere piccoli errori che portano via intere giornate per individuarli e correggerli, invece con la calma e la mente riposata basta poco tempo per analizzare, scomporre e risolvere anche grossi problemi.

I cicli di produzione devono essere corti. Al massimo lunghi qualche mese. All'interno di questi ci devono essere delle iterazioni molto brevi di alcuni giorni (al massimo una settimana). Questo favorisce il feedback e riduce il rischio di trovarsi a mezza via con la necessità di cambiare ed il software in uno stato non stabile e coerente.

Molto importante è la scelta di XP di eseguire prima solo i lavori con priorità più alta. Sono questi che hanno il maggior ritorno economico.

Cicli di produzione brevi significa poter richiedere all'utente di scegliere e proporre il minimo di funzionalità che rendano il progetto un affare.

Ogni ulteriore funzionalità può essere aggiunta nel successivo ciclo, partendo sempre da quelle che ha più senso introdurre.

Anche in caso di abbandono del progetto (richieste o priorità che cambiano) avere i cicli brevi assicura di perdere poco tempo e poco denaro. Cicli corti assicura anche che i cambiamenti durante un unico ciclo siano minimi e dunque sia raro abbandonare un progetto a metà.

## **2.2. Codice e test.**

Al centro di tutto c'è il codice. È con questo che si riesce a comunicare all'interno del gruppo di lavoro ed all'esterno. Il ciclo di vita ed il comportamento degli oggetti è definito nei test che non sono altro che codice. Per segnalare eventuali problemi si usano test che ne contengano una dimostrazione.

Un altro segnale dell'importanza del codice è l'applicazione del refactoring per migliorarlo continuamente.

La qualità del software prodotto da XP è alto: XP crea e mantiene test per ogni funzione e per ogni funzionalità per permettere di trovare velocemente nuovi errori introdotti. I test vengono eseguiti più volte al giorno ed in modo automatico.

I test permettono anche di comprendere meglio il codice, infatti in essi si possono facilmente trovare esempi di utilizzo.

Il codice deve essere chiaro ed autoesplicativo. I nomi utilizzati per gli oggetti ed i metodi devono permettere una comprensione immediata di ciò che rappresentino. Nomi come `timeCounter` ed `eventCounter` devono diventare la norma e sostituire gli abituali `c1` e `c2`.

Il codice chiaro e la presenza dei test autorizzano ad abbandonare la necessità di una documentazione dettagliata su tutto il software.

Spesso è molto più difficile spiegare a parole che interpretare alcune righe di codice: il codice non può essere ambiguo come il linguaggio comune.

Inoltre la documentazione tende facilmente ad essere disallineata al codice, che si modifica più velocemente. Solo la rigidità della programmazione nelle metodologie pesanti garantisce l'allineamento tra codice e documentazione. Non garantisce però che il processo sia veloce, che la documentazione sia comprensibile e che venga utilizzata.

Prima si fanno i test, poi il codice. Quando i test hanno successo, e non ci viene in mente nessun altro test che possa dare errore, il codice è pronto.

L'integrazione segue immediatamente lo sviluppo e questa comprende eseguire i test sull'intera applicazione. L'integrazione non è completata finché esiste un test che fallisce.

## **2.3. L'economia, il management ... l'uomo.**

XP non tiene conto solo dei processi legati alla pura produzione del software, ma considera il problema nella sua interezza partendo dalle motivazioni per cui si scrive software.

La programmazione deve essere considerata parte di una attività economica e dunque bisogna poter applicare i metodi dell'economia per calcolare la convenienza di un progetto: si deve stimare il valore attuale di ogni investimento pesandolo con il rischio ad esso associato. Bisogna sempre considerare tutte le eventualità e scegliere quella che probabilisticamente porta ad avere un maggiore valore attuale.

Queste in sintesi le opzioni che bisogna considerare:

- Innanzi tutto bisogna scegliere il momento in cui conviene intraprendere un nuovo progetto od aumentare gli investimenti se si

tratta di uno già avviato...

- Si deve considerare la possibilità di deferire una parte o la totalità degli investimenti in un momento futuro più opportuno.
- A volte può essere conveniente abbandonare un progetto e salvare solo alcune parti utilizzabile per nuovi progetti.
- Spesso bisogna cambiare la direzione e la strategia di un progetto per ottenere il massimo profitto.

Questi punti ricordano che le attività economiche possiedono un rischio intrinseco e che questo va controllato per la buona riuscita dell'impresa.

Per poter tenere sotto controllo il rischio, manager, utente e sviluppatore devono suddividersi i compiti continuando ad interagire.

Quattro sono le variabili che possono essere controllate: costo, tempo, qualità, prospettive (o campo d'azione).

Manager e utente possono controllare fino ad un massimo di tre variabili a scelta. La quarta deve essere lasciata al programmatore, altrimenti le richieste finiscono per diventare troppo pressanti, lo stress aumenta, si allungano i tempi e la qualità decade.

Per quanto riguarda il costo ed il tempo bisogna trovare una giusta misura perché il troppo, come il poco, possono avere effetti negativi.

È possibile sacrificare la qualità per avere dei guadagni a breve termine (in denaro o in tempo), ma il costo a lungo termine può rivelarsi enorme per quanto riguarda l'aspetto umano, quello tecnico e quello d'affari.

Un campo d'azione più piccolo permette di avere migliore qualità in tempi brevi e a basso costo.

Il legame fra queste quattro variabili è forte e complesso. Per esempio è insensato investire troppo all'inizio di un progetto sperando di coprire subito tutto il campo d'azione: il rischio è di ottenere solo disorganizzazione e aumento dei problemi da risolvere.

Invece quando il progetto è ben avviato, con più soldi, si possono avere più tecnici, macchine più veloci, organizzazione migliore, così da ridurre i

tempi di sviluppo, incrementare la qualità ed aumentare le prospettive.

Spesso il tempo è una variabile difficilmente controllabile: è il mercato stesso che impone dei vincoli.

La qualità è un aspetto che viene spesso sottovalutato e magari preso in considerazione solo nelle fasi finali del progetto: durante la messa a punto e la correzione degli errori.

Eppure avere un software qualitativamente migliore aiuta nella produzione e permette di restringere i tempi. Per favorire ciò sono utili i test: aumentano la confidenza dei programmatori con il codice diminuendo il loro stress e portando la qualità ad un certo livello standard. Il programmatore è contento di produrre software di buona qualità e questo ha un effetto positivo a lungo termine.

Poiché le specifiche di un progetto possono cambiare velocemente, il campo d'azione deve essere tenuto come una variabile malleabile per poter continuare a soddisfare le altre tre. Se per esempio i tempi si stringono, limitando il numero di caratteristiche che vogliamo ottenere possiamo arrivare al traguardo nei tempi prefissati mantenendo alta la qualità.

L'importante è realizzare prima le funzionalità fondamentali, così saranno solo quelle superflue ad essere lasciate indietro per la prossima release.

Per evitare di trovarsi alle strette è importante che il programmatore impari a fare stime sempre più accurate sui tempi di produzione.

XP richiede una stretta collaborazione fra utente e sviluppatore per evitare incomprensioni. Il progetto può così essere raffinato continuamente e riflettere le conoscenze acquisite man mano che la realizzazione del progetto procede. Le nuove richieste al cambiamento da parte dell'utente devono essere sempre ben accette.

Il gruppo di sviluppatori non deve nemmeno accorgersi se sta lavorando su una nuova funzionalità o su una già presente.

Ognuno è responsabile del proprio lavoro: è l'individuo stesso che riesce a fare migliori stime su quanto tempo gli occorra a completare un certo lavoro e gli individui vengono stimolati a fare delle stime sempre più precise

in base all'esperienza. Questo evita che venga richiesto l'impossibile ad un programmatore da parte di personale non addetto e non in grado di pesare adeguatamente la difficoltà di realizzazione. Tutto ciò evita che il programmatore diventi frustrato per non aver centrato gli obiettivi.

Il manager è anche responsabile di scegliere la tecnologia utilizzata, partendo comunque dai suggerimenti degli sviluppatori. Se questa scelta fosse lasciata agli sviluppatori si rischierebbe di utilizzare tecnologie vecchie a causa delle abitudini o al contrario tecnologie troppo nuove e instabili a causa della voglia di sperimentare. Il manager deve scegliere tenendo conto dei costi per acquisire la tecnologia e le competenze per utilizzarla, i costi per mantenerla e la possibile durata.

XP incoraggia il contatto umano tra membri del gruppo per ridurre il senso di solitudine che porta alla scontentezza sul lavoro.

Deve essere favorita la programmazione in coppia. Bisogna sempre essere pronti a rispondere positivamente ad una richiesta di collaborazione o di aiuto.

## **2.4. Il costo delle modifiche.**

Fino a qualche anno fa si aveva la corretta nozione che il costo per fare modifiche al software già scritto fosse circa esponenziale con il tempo.

Un problema che poteva facilmente essere corretto durante le prime fasi di sviluppo si rivelava costosissimo se il software era già entrato nella fase di produzione.

Una famosa frase utilizzata soprattutto dai responsabili software: "*codice che va non si tocca*" stava a significare che le modifiche si fanno solo se è veramente necessario ed è meglio *attaccare una pezza* in una piccola parte di codice che stravolgere tutto.

Sono stati compiuti enormi sforzi per cercare di ridurre il costo del

cambiamento e per aumentare la flessibilità.

- I linguaggi di programmazione vengono migliorati continuamente. Si pensi alla recente evoluzione verso una completa programmazione ad oggetti con forti spinte verso la modularità. I messaggi che gli oggetti si mandano sono ognuno una nuova opportunità per introdurre delle modifiche limitandosi a piccole parti di codice.
- Le metodologie di programmazione si evolvono grazie all'introduzione di nuove pratiche.
- Gli strumenti di sviluppo sono sempre più potenti e permettono controlli e azioni che facilitano le modifiche.
- La tecnologia che gira attorno ai database viene sempre più integrata nei linguaggi di programmazione e l'obiettivo è la completa trasparenza degli accessi ai database. Anche i database vengono visti ad alto livello come un insieme di oggetti, ognuno con i suoi metodi, e non come tabelle. Il codice è così attaccato ai dati e la migrazione diventa un processo poco costoso.
- Sono state introdotte nuove notazioni per migliorare la comprensione del codice che favoriscono la possibilità di modificarlo. Ricordiamoci che utilizzare un linguaggio ad oggetti non significa automaticamente programmare ad oggetti e sfruttarne tutte le potenzialità.

Grazie al progresso che c'è stato, la curva di crescita esponenziale dei costi del cambiamento non è più valida. Questo è il punto base da cui parte XP: senza questa premessa tecnica non ci si sarebbe nemmeno posti il problema di fare extreme programming.

Se precedentemente si tendeva a prendere tutte le decisioni nella prima fase della progettazione, ora è meglio posticiparle il più possibile e soprattutto quelle non necessarie. Una volta durante la fase di progettazione bisognava esaminare tutti i più piccoli particolari e pianificarli in anticipo senza magari neanche conoscerli bene.

Ora si implementa solo quello che serve in questo momento. Si aggiungono nuovi elementi o modifiche solo se semplificano il codice esistente o permettono di scrivere codice più semplice per la nuova funzionalità che ci apprestiamo ad introdurre.

I test automatici permettono inoltre di fare le modifiche senza il timore di danneggiare un codice funzionante o cambiare involontariamente il comportamento del sistema che stiamo costruendo. È importante abituarsi a variare spesso il codice, così quando le modifiche diventano inevitabili non ci spaventi l'idea di mettere le mani nel vecchio codice.

Tutti questi fattori contribuiscono ad appiattare la curva che rappresenta il costo delle modifiche durante il tempo. Una modifica fatta in un momento successivo continua a costare di più di una modifica fatta oggi, ma il divario è molto minore ed è accettabile.

D'altra parte si riesce a risparmiare tantissimo tempo concentrandoci sull'unica cosa che effettivamente ci interessa in questo momento senza essere obbligati a preoccuparsi e ad analizzare tutte le eventuali implicazioni che potranno sorgere in conseguenza di ogni decisione che prendiamo.

Molte volte si spreca tantissima energia per raffinare una parte di codice che alla fine viene scartata perché non più necessaria.

L'attenzione del programmatore viene spostata sul bisogno di **prendere decisioni velocemente** anziché prendere decisioni in modo che siano valide per tutta la durata del progetto.

## 2.5. I valori.

Spesso gli obiettivi a breve termine vanno in conflitto con gli obiettivi a lungo termine.

La società umana ha sviluppato un insieme di valori per evitare che gli interessi a breve termine di ogni individuo non coincidessero con quelli dell'intera umanità.

Per perseguire gli obiettivi a lungo termine anche XP necessita di alcuni valori da seguire.

Il primo di questi valori è la **comunicazione**.

Comunicare è fondamentale. Permette di migliorare la conoscenza del progetto, degli strumenti, dei metodi di lavoro, delle sensazioni e dei sentimenti dei compagni di lavoro. La maggior parte dei problemi che si incontrano in un progetto possono essere ricondotti ad una comunicazione non avvenuta.

Quando si comunica non bisogna aver timore di dare o sentire cattive notizie. Chi ascolta non deve giudicare, ma collaborare per risolvere il problema.

XP si propone di stimolare la comunicazione attuando pratiche che non possono farne senza. Sono tutte pratiche fondamentalmente a breve termine e dunque attuabili senza troppo sforzo di volontà.

Attuare pratiche come testing, programmazione in coppia e stima sui tempi implica necessariamente che ci sia comunicazione fra programmatori e utenti.

A volte anche tutto questo non basta, dunque è necessario che una persona sia addetta a controllare che il livello della comunicazione rimanga valido. Questa persona deve possedere particolari qualità nel notare e correggere quei piccoli comportamenti che tendono ad allontanare le persone.

Il secondo valore è la **semplicità**.

Siamo stati abituati a fare codice generico e generale per andare incontro a bisogni che potrebbero presentarsi nel futuro. Ma questo va bene se i

costi crescono esponenzialmente con il tempo, non per XP.

XP prende per buono che sia meglio fare la cosa più semplice possibile oggi e pagare un piccolo prezzo più avanti anziché realizzare cose complicate che magari non saranno mai utilizzate.

Comunicazione e semplicità hanno il pregio di supportarsi a vicenda: il dialogo permette di conoscere esattamente cosa deve essere fatto e cosa no. Più il sistema è semplice e meno si ha bisogno di comunicare.

Il terzo valore è la **retroazione** e può essere fatta a diversi livelli.

Nel breve tempo vengono scritti i test per controllare il codice che può portare a corruzioni. I programmatori possono così avere un feedback continuo su quello che scrivono e sullo stato del sistema.

Quando i direttori propongono nuove storie (scomposizioni di nuove funzionalità) ricevono subito un feedback dai programmatori che stimano i tempi di realizzazione.

Viene tracciato il completamento dei processi di programmazione per un controllo a medio termine.

Il feedback è presente anche a lungo termine: vengono fatti test funzionali su casi d'uso semplificati. Viene tenuto sotto controllo dal direttore l'andamento generale del lavoro per correggere la pianificazione dei tempi.

All'inizio di un progetto ci si concentra sugli aspetti più importanti per arrivare velocemente ad una prima piccola release che ci dia subito una risposta di retroazione.

A parte questa piccola fase iniziale lo sviluppo del software deve viaggiare in parallelo alla finalizzazione di successive release del progetto; bisogna imparare a supportare la produzione mentre si continua ad introdurre nuove funzionalità (partendo sempre da quelle che hanno più senso).

Questo è anche ciò che oggi viene richiesto dal mercato che è sempre più volubile.

Il feedback aumenta con la facilità di comunicare: anziché fare

un'obiezione si può scrivere un test che dimostri se c'è qualcosa che non va.

Il quarto valore è il **coraggio**.

Se ci si accorge di stare seguendo una strada sbagliata bisogna avere il coraggio di ritornare sui propri passi e ricominciare dal punto giusto.

Alla fine vedere che tutti i test hanno successo premia le nostre scelte e ci infonde nuovo coraggio.

Se si fanno delle modifiche ed i test che prima passavano ora falliscono miseramente bisogna avere il coraggio di gettare via il nuovo codice, oppure comprendere che il sistema ha bisogno di modifiche più profonde.

Se si è indecisi tra più strade si può provarle tutte ed alla fine scegliere la più promettente.

Il coraggio permette di trovare una via d'uscita anche quando sembra che tutte le strade siano a fondo cieco. Si inizia pian piano a modificare qualcosa per poter guardare il problema da prospettive diverse tentando di trovare una pista che ci chiarisca il nostro problema.

Il coraggio da solo non basta. Senza i primi tre valori il coraggio si riduce ad incoscienza.

La comunicazione supporta notevolmente il coraggio: in due o più persone si ha meno paura di commettere un grave errore di prospettiva.

Viceversa per comunicare bisogna avere il coraggio di riferire le buone e le cattive notizie, di ammettere i propri errori ed i propri limiti, confidando di poter risolvere ogni difficoltà sfruttando le capacità di tutti.

Un sistema semplice e più comprensibile aiuta ad essere più coraggiosi e viceversa tramite il coraggio è possibile fare modifiche che semplifichino il sistema.

La retroazione concreta infonde coraggio, così come bisogna avere coraggio le prime volte che si iniziano a scrivere i test e sembra di gettare inutilmente del tempo che potrebbe essere usato per creare codice per il programma.

Alla base di questi quattro valori ce ne è un altro: il **rispetto**.

Rispettare le varie scelte fatte dai vari membri del gruppo è fondamentale. XP non può sopravvivere in un ambiente in cui l'individualismo prevale.

Se il gruppo prende a cuore il progetto, XP subito innesca un feedback positivo e permette di essere felici di fare parte di un progetto e di un gruppo che ha successo.

## 2.6. I principi.

Dai valori descritti nel precedente paragrafo XP deriva diversi principi pratici che aiutano a scegliere tra diverse alternative che ogni volta ci si prospettano. Ogni principio incorpora i valori trasformandoli in qualcosa di concreto.

Una **rapida retroazione** (feedback) è alla base dell'apprendimento.

Dobbiamo imparare a risolvere i problemi con **semplicità**. Ci è stato insegnato a pianificare il futuro ed a progettare software per facilitare il riutilizzo di ogni componente. XP al contrario incentiva a risolvere i problemi attuali e ad avere fiducia di essere all'altezza in futuro di aggiungere complessità.

Questa complessità può poi essere raggiunta tramite **modifiche incrementali**. Una grossa modifica può essere fatta per piccoli passi, partendo da quelli più importanti.

Quando dobbiamo scegliere, prendiamo la strada che ci lascia più possibilità aperte per il futuro risolvendo il maggior numero di problemi attuali. **Abbracciamo il cambiamento** in modo che diventi un'abitudine.

Un programmatore estremo **viaggia leggero** portando avanti solo poche cose alla volta, sempre pronto a modificare la rotta. Chi tiene traccia del

cammino percorso sono il codice ed i test.

Utile dare delle **misure corrette** alle aspettative ed ai risultati. Non esagerare in precisione sulle previsioni, quando è noto come possono essere volubili.

Mai trascurare la **qualità**, perché tutti amano riuscire ad ottenere risultati qualitativamente validi.

Oltre a queste pratiche principali se ne possono aggiungere altre.

È difficile indicare quanto sforzo deve essere dedicato ai test, quanto al refactoring, ecc. Molto meglio **insegnare ad imparare** sperimentando.

**Piccoli investimenti iniziali** obbligano programmatori ed utenti a scegliere le specifiche e l'approccio migliori. Il troppo porta alla rovina, così come il troppo poco non permette di progredire.

Bisogna sempre **giocare per vincere** e non solamente per non perdere. Scrivere pagine e pagine di documentazione significa avere paura che il nostro codice sia poco comprensibile; incontri lunghi e frequenti nascondono la paura di essere fuori strada.

Ogni volta che si deve prendere una decisione bisogna fare degli **esperimenti concreti**. I risultati di una fase di progetto non deve essere un disegno finito, ma una serie di esperimenti che mostrino le varie difficoltà incontrate e che indichino possibili strade su come risolverli.

Il risultato di una discussione sulle specifiche dovrebbe essere una serie di esperimenti. Ogni decisione astratta dovrebbe essere resa concreta tramite test.

I programmatori devono essere messi nelle condizioni di poter **comunicare apertamente** i problemi e di ricevere supporto e non punizioni.

Quando aumenta la pressione e lo stress le uniche pratiche che vengono seguite da uno sviluppatore sono quelle che risolvono problemi immediati.

Per questo le pratiche di XP che danno risultati a lungo termine **assecondano gli istinti delle persone** e promettono risultati anche nel breve termine.

Il programmatore deve essere capace di **accettare le responsabilità** che gli si presentano, autonomamente, senza essere costretto dall'alto, incrementando così le proprie motivazioni.

XP deve naturalmente essere **adattata** caso per caso alle singole esigenze. Niente deve essere preso come vero in assoluto.

## 2.7. Le quattro attività essenziali.

**Scrivere codice** è sicuramente l'attività più importante. Il risultato del nostro lavoro è un programma, cioè codice.

Ogni volta che abbiamo un'idea, la miglior prova per vedere se è valida è trasformarla in codice. Scrivere codice ci permette anche di sviluppare l'idea ed imparare.

Il codice ci permette anche di comunicare le idee senza ambiguità. Programmando assieme si possono facilmente comprendere le idee della persona accanto osservando mentre digita.

Quello che si osserva sono le idee che prendono forma ed ognuno nella propria mente le può rielaborare e comprendere in maniera personale.

Questo tipo di stretta comunicazione è il miglior modo per sviluppare le idee e per crearne altre.

La duplice forma che le idee prendono nella nostra mente e nel codice può facilmente divergere.

**Scrivere test** ci permette di dare forma all'idea in una maniera indipendente da come l'abbiamo implementata (o la implementeremo) nel codice. Questo ci permette di controllare che abbiamo scritto proprio ciò che volevamo o ciò che ci era richiesto.

Infatti i test possono essere scritti dai programmatori per testare l'implementazione del codice (unit test), oppure dagli utenti o secondo le specifiche degli utenti per testare le funzionalità.

Questi ultimi hanno anche il pregio di dimostrare agli utenti che quello prodotto è un software valido che risponde alle loro esigenze.

Si possono fare test anche per controllare altri parametri non direttamente collegati con il codice come per esempio le performance.

Scrivere i test deve essere posta tra le attività essenziali di un programmatore anche per altri due motivi:

- Il primo è che i test permettono al codice di sopravvivere a lungo nonostante le continue modifiche.
- Il secondo motivo è anche quello che garantisce che i test vengano effettivamente scritti senza bisogno che qualcuno ci costringa a farlo:  
programmare scrivendo test è più divertente e più soddisfacente che programmare senza test.

Grazie ai test è possibile programmare con molta più sicurezza e senza timori di danneggiare codice funzionante ogni volta che si introduce una modifica.

Vedere che tutti i test passano significa essere continuamente rassicurati ed appagati.

La cosa più stupefacente è che programmare scrivendo test è anche più veloce che programmare senza.

All'inizio ciò non è vero poiché serve del tempo per abituarsi all'idea dei test e per imparare come costruirli, poi il guadagno in produttività deriva dal tempo che si risparmia evitando le fasi di debugging.

Allo stesso tempo fare cattivi test può portarci ad avere una falsa confidenza.

Per sapere cosa deve essere codificato e testato i programmatori devono **comunicare** con gli utenti e comprendere i loro problemi e le loro esigenze. Allo stesso tempo devono far sapere loro cosa può essere codificato facilmente e cosa necessita più attenzione, evitando di comunicargli dettagli inutili.

Codificare solamente non è sufficiente. Più la quantità di codice ed i test aumentano, più la complessità aumenta. Prima o poi si arriva ad un punto in cui ci vediamo costretti a mettere mano ad una gran parte del codice per riuscire a fare anche solo una piccola modifica.

Ciò è inevitabile.

Se il codice scritto è un buon codice questo momento può essere ritardato, ma prima o poi arriva. A questo punto serve **ridisegnare** il codice, creare strutture per riorganizzare la logica.

Si cerca di fare in modo di favorire la localizzazione delle modifiche. Ogni pezzo di logica non deve essere duplicato nel codice, né sparso.

Deve inoltre essere messo accanto ai dati su cui opera.

Questa fase di progetto XP la fonde con la scrittura del codice. Non sono due fasi separate, ma due aspetti della stessa. Progettare fa parte dell'attività quotidiana di chi scrive codice.

È il codice stesso che ci indica quando è il momento di ristrutturarlo. Se ci accorgiamo che stiamo duplicando più volte il codice, oppure se facciamo fatica a comprendere o a ricordarci come funziona un certo oggetto, allora è il momento.

## **3. Applicare programmazione estrema.**

### **3.1. Le tecniche.**

Le quattro principali attività vengono applicate da XP tramite dodici tecniche, dette anche pratiche.

#### **1. Veloce pianificazione.**

I requisiti sono raccolti dall'utente, per tutta la durata del progetto, tramite storie di funzionamento del sistema. Il contatto costante con l'utente è necessario durante tutto lo sviluppo. Una storia è un sintetico caso d'uso.

Il manager determina velocemente le finalità del prossimo ciclo di consegna combinando le priorità del commercio e le stime tecniche dei programmatori. Tali piani vengono modificati se le condizioni esterne cambiano.

Le persone addette agli affari ed al commercio devono prendere le decisioni su cosa bisogna puntare l'obiettivo. Devono scegliere quali siano le storie prioritarie, quali da scartare e le date di consegna. Queste decisioni non possono essere prese dai programmatori perché per farle bisogna conoscere le esigenze del mercato.

Per prendere queste decisioni gli uomini d'affari devono però basarsi sui consigli e le considerazioni tecniche degli sviluppatori.

Questi ultimi devono saper fare delle stime sui tempi di realizzazione, informare gli uomini d'affari sulle conseguenze tecniche che certe decisioni possono avere.

I programmatori devono potersi organizzare il proprio lavoro autonomamente perché sono loro che conoscono i problemi tecnici.

Gli uomini d'affari fissano le priorità che devono essere soddisfatte al termine di una release, ma all'interno di una release sono i programmatori che scelgono liberamente di provare a scrivere per primi i processi più critici per riuscire a ridurre il rischio globale, sempre tenendo conto delle priorità fissate.

## **2. Piccoli cicli di consegna.**

Ogni ciclo deve essere il più corto possibile e deve rispondere alle esigenze che portano maggior valore al progetto. Si riesce così a mantenere basso il rischio legato ad una modifica delle specifiche o all'abbandono del progetto.

## **3. La metafora.**

Per metafora si intende una architettura che sia facile da comunicare e da elaborare. Serve per far comprendere a ciascuno gli elementi base che deve conoscere e le relazioni tra tali elementi.

I termini per indicare gli oggetti devono essere presi dalla metafora. La metafora non è un progetto finale, ma è in continua evoluzione e cresce assieme allo sviluppo del software. Il suo unico scopo è assegnare a ciascuno una storia all'interno del quale possa lavorare.

## **4. Semplici propositi.**

Avere semplici intenzioni significa seguire poche regole. Devono passare tutti i test. Non deve esserci codice duplicato, in nessun caso.

Il codice deve riuscire a segnalare tutte le intenzioni dei programmatori. Deve contenere il minor numero possibile di classi e di metodi: solo le cose necessarie devono esserci.

Il sistema deve essere sempre realizzato nella maniera più semplice e la complessità va subito rimossa se non è necessaria.

## **5. I test.**

Per i programmatori test sul codice; per gli utenti test funzionali. Per entrambi aumenta la confidenza che il lavoro è ben fatto.

Se un programmatore vuole esplorare una nuova strada per vedere se funziona può evitare di scrivere i test: se la strada è da scartare verrebbe sprecato il tempo impiegato per costruire i test. Però appena si decide il percorso giusto bisogna partire subito dai test.

## **6. Ristrutturazione.**

È quasi impossibile riuscire a costruire immediatamente un software ben organizzato in cui ogni oggetto sia indipendente dagli altri, astragga un solo concetto e contenga solo i metodi che agiscano sui propri dati. Appena la complessità aumenta o le specifiche cambiano, anche se si spendessero ore ed ore di pianificazione prima di scrivere il codice, comunque questo necessiterebbe di essere riorganizzato.

Quando aggiungiamo nuovo codice dobbiamo sempre chiederci se ci sia un modo migliore per riorganizzare il programma in modo che sia più facile introdurre le modifiche.

È inutile perdere troppo tempo per progettare una struttura completa prima di iniziare a scrivere codice. Al contrario è importante riorganizzare il codice alla fine chiedendosi quali possibilità ci sono per semplificarlo. Questo ci permette di avere un programma sempre pronto ad accettare nuove modifiche.

Per chi si mostra riluttante a modificare codice già funzionante ci sono alcune valide argomentazioni che possono farlo ricredere:

- i test ci assicurano che alla fine tutto sarà corretto
- la ristrutturazione può essere fatta a piccoli passi, abbassando così il rischio di perdere la testa
- programmando in coppia è meno probabile commettere errori per una svista.

Bisogna stare molto attenti a non fraintendere il significato di refactoring. Non significa prevedere le modifiche che apporteremo in futuro. Noi riorganizziamo il codice solo per renderlo più **semplice**, **comprensibile**, **coerente** e **flessibile**, evitando di avere codice duplicato.

## **7. Programmazione in coppia.**

Seduti di fronte ad un computer devono esserci sempre almeno due persone di cui una ha il controllo della tastiera ed il mouse e svolge lo stesso lavoro che farebbe se fosse da sola. L'altra ha il compito di considerare il problema da un punto di vista più elevato: deve pensare se strategicamente si stia implementando la soluzione corretta, se ce ne sia un'altra più semplice.

Quando si digita spesso è facile farsi trascinare dalla voglia di arrivare in fondo alla soluzione troppo in fretta. In due è più facile controllarsi a vicenda, evitando di trovarsi alla fine a passare lunghe ore in fase di debugging, magari per trovare subdoli errori come la mancata delocalizzazione di un'area di memoria.

## **8. Proprietà collettiva del codice.**

All'inizio della storia della programmazione si permetteva di modificare il codice a chiunque. Spesso però le modifiche non erano coerenti con altre parti del programma ed alla fine si cadeva nel caos. Numerosi progetti sono falliti dopo una serie di tentativi di correggere errori con il solo risultato di aggiungerne altri.

Per ovviare a ciò venne introdotto il principio della proprietà. Chi costruiva il codice era anche responsabile di mantenerlo. A lui dovevano essere richieste le modifiche. Si sono così risolti parecchi problemi al prezzo di rendere più lento lo sviluppo: le modifiche richieste non venivano fatte istantaneamente.

XP vuole fare in modo che ognuno sia responsabile dell'intero sistema. Appena si trova un punto in cui è necessario intervenire, si procede.

Altre tecniche come i test o la programmazione in coppia ci danno la fiducia necessaria e ci garantiscono di non introdurre bachi, evitando di ritornare al caos dovuto all'anarchia.

## **9. Continua integrazione.**

Le modifiche al codice vengono fatte in locale, così ognuno può lavorare anche su parti di codice su cui stanno lavorando altri. Per evitare conflitti irrisolvibili è però necessario integrare spesso il codice scritto: appena un processo è terminato, dopo poche ore o giorni, non settimane.

L'ideale è avere un computer libero su cui far girare tutti i test prima di integrare le modifiche. Finché i test non passano tutti, la modifica non può essere accettata. Viene così assegnata inequivocabilmente la responsabilità di chi debba correggere i test che non passano.

## **10. 40 ore settimanali.**

Questo può sembrare inaccettabile oggi in un mondo in cui ci sarebbe richiesto di essere presenti almeno 24 ore al giorno (?!).

Eppure programmare richiede di essere concentrati e motivati costantemente.

È impossibile che non capiti una settimana in cui si possa proprio evitare di fare del lavoro straordinario. Però se la settimana successiva ci sentiamo ancora alle strette significa che esiste una incertezza di fondo e che siamo mal organizzati.

## **11. Utente sempre presente.**

Avere sempre a disposizione un utente a cui rivolgere domande permette di costruire un software rispondente alle esigenze. Un utente che segue lo sviluppo del prodotto può facilmente accorgersi di problemi o di nuove specifiche che possono essere introdotte per migliorare il sistema.

Il tempo speso dall'utente non viene sicuramente sprecato. Inoltre non è detto che non riesca anche a seguire il suo lavoro personale.

Spesso il tempo da dedicare agli sviluppatori è poco; quello che fa la differenza è la tempestività con cui le risposte arrivano.

## **12. Regole di codifica standard.**

Tutti devono adottare le stesse regole per codificare, altrimenti ogni volta che si modifica una parte di codice scritta da un altro si è costretti a correggere quelle che per noi sembrano imperfezioni.

Con un po' di attenzione si può facilmente arrivare a scrivere codice perfettamente uniforme e che è accettato volontariamente da chiunque.

La cosa fondamentale è evitare che ci sia codice duplicato. Inoltre lo standard deve permettere di impiegare la minore fatica possibile.

Un ultimo aspetto: lo standard deve favorire la comprensione del codice e la comunicazione.

## 3.2. Mettendo tutto assieme.

I migliori risultati si ottengono dall'unione di tutte queste dodici tecniche. Nel capitolo precedente già ho accennato a come alcune tecniche non possano funzionare senza la contemporanea presenza di altre.

La tecnica di costruire i test automatici può essere valida anche in assoluto, però per le altre gli aspetti negativi spesso contano di più dei vantaggi che portano, a meno che le debolezze di una pratica non vengano evitate grazie alla forza delle altre.

Molte di queste tecniche sono state scartate nel passato perché inefficienti. Assieme però possono contribuire ad abbassare la curva esponenziale del costo delle modifiche nel tempo. A sua volta questo effetto permette di considerare la programmazione da un altro punto di vista e rende ancora più attuali le pratiche descritte.

La metodologia vincente non è più una metodologia predittiva che gioca in difesa, ma una metodologia adattativa.

Vediamo alcuni esempi di come le varie tecniche riescano a supportarsi a vicenda.

Per potersi permettere di programmare partendo da una pianificazione semplice e creata in breve tempo è importante avere piccoli cicli di consegna in modo da poter subito correggere errori.

Avere un utente sempre presente permette di rilevare tali errori il più velocemente possibile.

Piccoli cicli di consegna possono essere fatti se integriamo spesso le modifiche nell'applicazione facendo sempre passare tutti i test.

Non serve così una fase di testing pre-release. Avendo uno stile semplice e realizzando sempre prima le funzionalità più importanti è possibile consegnare un prototipo funzionante già dopo pochi giorni di programmazione.

Per evitare di pensare ad una completa architettura bisogna avere una metafora comprensibile a tutti, compresi gli utenti. Serve una retroazione veloce per controllare che la metafora sia costantemente valida, ristrutturando continuamente il codice per meglio adattarla e raffinarla.

Un semplice proposito significa essere pronto a ritornare sui propri passi per modificare quello che non avevamo prima previsto.

Ristrutturare deve essere una abitudine, in modo che non ci spaventi l'idea di dover modificare il codice già scritto.

La metafora deve essere chiara per evitare che i futuri propositi non siano corretti. Programmare in due ci assicura di non commettere errori quando scegliamo i propositi su cui basarci.

Per scrivere test senza perdere troppo tempo bisogna costruire il software a piccoli passi seguendo semplici propositi. In tal modo sarà possibile testare ogni piccolo passo senza difficoltà. La programmazione in coppia ci assicura che i test non vengano sottovalutati o abbandonati.

Una cosa da sottolineare è l'estrema importanza che hanno i test per XP e per tutte le altre tecniche. Per esempio la ristrutturazione sarebbe impossibile senza i test che ci assicurino di non introdurre difetti.

Ristrutturare diventa una attività si routine se assieme ai test abbiamo anche la programmazione in coppia che ci infonde coraggio, semplici propositi, regole standard di codifica, continua integrazione per assicurarci di poter facilmente tornare indietro in caso di errore, mente fresca e riposata per non commettere banali errori.

Per programmare in coppia è necessario essere d'accordo sulle regole di codifica. Programmando in coppia il livello di concentrazione e di attenzione rimane costantemente alto, dunque non è possibile lavorare in tali condizioni per dieci ore al giorno, tutti i giorni. La metafora serve per evitare di comunicare senza comprendersi. I test vengono possibilmente fatti prima del codice per avere entrambi le idee più chiare quando si inizia a scrivere.

Esercitare la proprietà collettiva del codice necessita frequente integrazione per rendere disponibili subito a tutti le modifiche introdotte. I test ci assicurano di non modificare erroneamente codice scritto da altri. Codice il cui funzionamento può meglio essere compreso lavorando in due. È necessario utilizzare tutti le stesse regole di codifica.

Per integrare spesso bisogna avere i test sempre efficienti. Ristrutturare permette di avere il codice scomposto in tante piccole unità in modo da ridurre la probabilità di conflitti.

Lavorare 40 ore alla settimana diventa possibile quando tutte le altre pratiche vengono applicate e la metodologia XP ci assicura di avere un processo efficiente e di qualità. Si riduce così anche la frequenza dei problemi inaspettati che ci costringono a lavorare oltre l'orario.

Un utente sempre presente può produrre valore segnalando le priorità e scrivendo test funzionali per indicare i suoi desideri.

Le 12 tecniche utilizzate da XP possono essere grossolanamente divise in 2 gruppi:

- 1) Tecniche che dipendono dall'**individuo** e che possono essere anche applicate in un gruppo che non aderisce all'extreme programming: testing, ristrutturazione, propositi semplici, ed in un certo modo anche la programmazione in coppia, lavoro settimanale non eccessivo e regole standard di codifica.
- 2) Tecniche che dipendono dall'intero **gruppo** di lavoro: piccoli cicli di consegna, continua integrazione, veloce pianificazione, utente (cliente) sempre presente sul posto di lavoro, proprietà comune del codice, la metafora.

### 3.3. XP per i manager.

XP, partendo dai principi visti a nel paragrafo 2.6, costruisce anche una modalità di comportamento adatta ad un manager a capo di un gruppo di programmatori estremi.

Il compito di un manager non è quello di comandare e decidere, ma quello di evidenziare quali sono i bisogni dell'azienda. Il lavoro non deve essere assegnato ad ogni singolo programmatore, ma sono gli sviluppatori stessi che devono assumere ognuno le proprie responsabilità e scegliere i lavori spartendosi quelli che il manager indica come prioritari.

Allo stesso modo non serve a nulla fornire manuali di comportamento ai programmatori (la maggior parte delle volte non vengono nemmeno letti), bisogna semplicemente seguirli e guidarli quando hanno bisogno.

In fondo i programmatori sono sempre propensi a realizzare lavori in cui prevalga la qualità.

Non devono essere inoltre imposte ai programmatori lunghi lavori che non riguardino le 12 pratiche come ad esempio scrivere dettagliate ed inutili documentazioni sul software prodotto, lunghe analisi sullo stato del sistema, incontri estenuanti ed improduttivi.

Spetta al manager riuscire ad adattare la metodologia di lavoro (magari proprio XP) alle attuali condizioni di lavoro che si trovano nell'azienda.

Il manager deve occuparsi di **misurare** le prestazioni del gruppo di sviluppo confrontandole con le aspettative. Grazie a questa misure è possibile accorgersi se qualcosa non va. Se per esempio il ritmo di produzione aumenta significa che XP viene correttamente implementata e che si vedono i frutti.

Il motivo però può anche essere che si stiano trascurando alcune pratiche come la ristrutturazione e la programmazione in coppia e dunque bisogna

indagare e capire le ragioni altrimenti si rischia di pagare salato in futuro alcuni piccoli guadagni presenti.

Le misure devono poter essere fatte semplicemente e senza disturbare troppo il lavoro degli sviluppatori.

Lo scopo di queste misure, oltre ad avere un controllo sulla situazione, è riuscire a presentare un importante feedback ai programmatori. Per questo motivo sono sufficienti poche e semplici misure e devono essere presentate solo quelle che mostrano una carenza in modo da attirare l'attenzione su tale aspetto. Quando poi il problema si sistema si può smettere di insistere su tale misurazione e magari iniziare con altre.

Il manager deve utilizzare queste misure per far comprendere in maniera gentile quali sono i problemi che vorrebbe vedere risolti.

All'interno della figura di manager che è stata descritta si possono individuare due distinti compiti che possono essere assunti anche da persone diverse.

Il primo compito è proprio quello di **misurare** le differenze tra le stime ed il tempo realmente impiegato e realizzare le funzionalità.

Il secondo compito è quello di **assistere** i programmatori. Non è necessario essere degli sviluppatori provetti, né degli esperti di architettura dei sistemi. Le qualità più importanti sono quelle legati ai rapporti umani, alla comunicazione. Questo tipo di manager può essere impersonato da uno dei programmatori, ma deve essere sempre pronto a lavorare in coppia con chiunque abbia bisogno e per questo motivo a lui non devono essere assegnati altri processi di responsabilità che lo distoglierebbero dal suo compito.

Spetta a lui tenere fare da tramite tra i programmatori e gli altri manager, aiutando i primi su tutto ciò che riguarda la programmazione (sviluppare le abilità tecniche individuali) e rendere partecipi i secondi dello stato attuale del sistema e dei suoi problemi.

Un suo compito è fare in modo che non manchi mai ai programmatori del cibo da sgranocchiare durante le pause per riposare. Può diventare anche un modo per rafforzare i contatti umani tra i programmatori.

Alla fine un manager è anche costretto ad intervenire decisamente per risolvere problemi che potrebbero accrescere e diventare pericolosi per la stessa sopravvivenza dell'azienda. Anche in questi casi la miglior qualità del manager è l'umiltà.

Non serve incolpare nessuno per quello che sta succedendo, ma bisogna rivolgere gli sforzi di tutti per risolvere le difficoltà.

Esistono anche decisioni più gravose che devono essere prese come quella di abbandonare un progetto se non è più remunerativo e se esiste un progetto migliore a cui dedicarsi.

Se c'è qualcosa che non va il manager può trasmettere le sue sensazioni comunicandogli la necessità di cambiare. Saranno poi gli sviluppatori a decidere cosa sperimentare per migliorare. Compito del manager è fare delle misurazioni per retroazionare i risultati ottenuti con tali esperimenti.

### **3.4. L'ambiente.**

Nella teoria di XP vengono presi in considerazione anche ai dettagli, come per esempio la disposizione dei posti di lavoro.

Bisogna fare in modo che sia possibile lavorare in coppia di fronte ad un terminale.

Inoltre il team deve essere a stretto contatto e non sparso per vari uffici: bisogna favorire la comunicazione in ogni modo. Possono essere utili anche angoli dove ogni programmatore possa avere i suoi momenti di riservatezza, quando necessario.

Al centro dell'ambiente di lavoro va messa la macchina più potente, solitamente usata per fare i test e per integrare le modifiche.

È necessario avere anche un tavolo dove sedersi tutti assieme per tenere brevi riunioni.

Sono i programmatori che decidono come disporsi e come organizzarsi. La possibilità di prendere possesso dell'ambiente di lavoro e di renderlo efficiente al massimo può insegnar loro che le scelte fatte possono influenzare la riuscita del progetto.

## 4. Organizzazione del lavoro.

Ora che si conoscono le pratiche che devono essere applicate, bisogna imparare a metterle assieme.

Tutto il lavoro può essere scomposto in piccole parti che possono facilmente essere controllate. Per far ciò si inizia dalla **pianificazione strategica** tramite la quale arriviamo a costruire delle **storie** indipendenti che catturano i vari pezzi. Lo sviluppatore a sua volta suddivide le storie in processi semplici tramite la pianificazione iterativa.

### 4.1. Le pianificazione strategica.

Pianificare significa riunire la squadra, decidere gli obiettivi, assegnare delle priorità. Successivamente si stimano i costi e si programma la tabella di marcia. È importante assicurare tutti i membri della squadra che l'intero processo porti al successo.

Un ultimo punto, non meno importante, è fornire banco di prova per assicurare una giusta retroazione che renda tutti consapevoli dello stato del sistema.

Partendo dai principi base che avevamo visto nel paragrafo 2.6 si possono trarre velocemente alcune linee guida.

La pianificazione deve essenzialmente riguardare un periodo di tempo breve. È certamente significativo fare anche delle previsioni a lungo termine per sapere a grandi linee dove si vuole arrivare, ma livello dei dettagli deve rimanere scarso.

Soprattutto tali analisi non devono essere troppo dispendiose in modo che sia facilmente possibile correggerle ed aggiornarle.

Quello che ci interessa maggiormente è pianificare fino alla prossima distribuzione del prodotto o fino al termine del ciclo se il programma non deve essere immesso direttamente sul mercato.

La pianificazione è un processo che in XP viene svolto assieme da uomini d'affari e programmatori. Ognuno deve essere capace di assumersi le proprie responsabilità e nessuno deve imporre le proprie decisioni.

Questo prende maggior forza nel fatto che chi si assume le responsabilità è anche colui che può fare una stima sul tempo di realizzazione.

## 4.2. Le regole.

Uomini d'affari e sviluppatori devono avere fiducia gli uni degli altri, ma per ottenere ciò servono alcune semplici regole che permettano di mantenere le promesse anche quando il livello di stress aumenta e gli istinti individuali tendano a prevalere.

Tali regole servono solo per imparare uno stile di comportamento e possono anche essere rimosse quando le passioni sono sotto controllo e le corrette relazioni umane diventano una abitudine.

Lo **scopo finale** è massimizzare il valore del software prodotto dalla squadra, tenendo conto dei costi e dei rischi.

Per fare ciò si deve investire il minimo ed arrivare a produrre nel più breve tempo possibile, sempre continuando ad applicare le tecniche che ci assicurano di avere software di qualità e rischi minori.

Gli uomini d'affari possono subito trarre degli insegnamenti dai risultati conseguiti con le prime distribuzioni e concentrare il lavoro degli sviluppatori sulle funzionalità più importanti e più remunerative.

Tre fasi caratterizzano la pianificazione. Si parte da una fase di esplorazione in cui gli uomini d'affari scrivono delle storie che descrivono cosa il sistema deve fare. Queste storie devono essere abbastanza

semplici ed occupare al massimo un paragrafo ciascuna in modo da essere chiaramente leggibili e comprensibili.

A questo punto tocca agli sviluppatori stimare quanto tempo occorra per realizzare ogni storia. Se la stima si presenta complessa occorre chiedere a chi l'ha scritta di spiegarla meglio, oppure di suddividerla in storie più piccole.

In base a queste stime i responsabili degli affari possono scegliere la durata del presente ciclo e le funzionalità da implementare.

Per far ciò le storie vengono divise in tre gruppi:

- quelle essenziali per il funzionamento del sistema;
- quelle non essenziali, ma di alto valore commerciale;
- quelle semplicemente attraenti a livello commerciale.

Allo stesso modo gli sviluppatori dispongono le stesse storie in altri tre gruppi:

- quelle che si riescono a stimare facilmente;
- quelle più difficili da stimare;
- quelle che presentano un alto rischio tecnico e che non possono essere stimate affatto alla luce delle attuali conoscenze.

Gli sviluppatori indicano quali sono secondo loro i tempi di produzione in condizioni ideali. Gli uomini d'affari a questo punto scelgono le storie che devono essere implementate ed in accordo a ciò la data della consegna.

Durante lo sviluppo del progetto le conoscenze dell'intero gruppo aumentano e permettono di migliorare le stime e di correggersi.

Tipicamente un ciclo dovrebbe durare tre settimane e durante la prima di queste sarebbe opportuno realizzare un sistema completo, anche se rozzo, per controllare così la velocità di realizzazione.

Gli sviluppatori possono chiedere di ridurre il numero di storie da implementare se si accorgono che le loro previsioni sulla velocità sono state troppo alte.

Gli uomini d'affari possono aggiungere nuove storie se il mercato lo richiede. Tali storie andranno a sostituire quelle meno importanti.

Se ci si accorge che i piani distano in maniera elevata dalla pratica gli sviluppatori possono chiedere di rifare la stima delle storie ancora da realizzare.

Questo tipo di pianificazione permette agli uomini d'affari di intervenire ogni tre settimane per correggere la rotta dell'azienda e se necessario agire pesantemente.

### **4.3. Le pianificazione iterativa.**

All'interno del ciclo di tre settimane gli sviluppatori realizzano le storie scomponendole in **processi**, attuando così quella che viene chiamata pianificazione iterativa.

Nella fase di esplorazione i programmatori prendono le storie e le suddividono in processi che richiedano al massimo un paio di giorni di lavoro.

Anche in questo caso per descrivere un processo, come per le storie, devono essere sufficienti alcune righe. Un processo deve aver la proprietà di essere facilmente stimabile.

Finita questa fase, ogni programmatore accetta la responsabilità per un processo. Questo implica la sua stima in base al numero di ore per completarlo in condizioni ideali.

Con questo si intende il tempo che si impiegherebbe se non venissimo mai disturbati e se il lavoro filasse via sempre liscio.

Dall'altro lato deve essere calcolato anche il *fattore di carico*, ovvero la percentuale di tempo che viene effettivamente spesa a programmare, escludendo il tempo perso nella programmazione in coppia (un processo è assegnato solo ad uno dei due membri della coppia), alle riunioni, discutendo con gli utenti, ecc.

Il fattore di carico non è uguale per tutti: dipende dall'abilità del singolo programmatore e dai diversi impegni che un individuo può avere all'interno dell'azienda.

Potrà inizialmente stupire, ma il fattore di carico non potrà mai essere superiore a 8/15, cioè otto giorni di lavoro effettivi in tre settimane. Se ciò non fosse, sarebbe segno di problemi all'interno del gruppo: qualcuno non starebbe aiutando abbastanza i compagni.

Moltiplicando il fattore di carico per i giorni ideali stimati per completare i processi presi in consegna si ottiene il tempo totale. Se si hanno troppi processi a carico, alcuni processi devono essere lasciati.

I processi devono essere bilanciati e distribuiti equamente all'interno del gruppo di sviluppatori.

A questo punto alcuni programmatori iniziano a svolgere un processo, mentre altri diventano il secondo elemento della coppia. Poi i ruoli si possono invertire. Si inizia scrivendo i test per il processo, poi il codice finché i test non passano. Infine si eseguono tutti i test che esistono per l'applicazione per poter così integrare le novità.

Bisogna registrare i progressi fatti ogni manciata di giorni per conoscere lo stato del sistema. Questo lavoro può essere gestito da uno dei membri della squadra.

Se un programmatore è indietro rispetto ai piani, può ridurre gli obiettivi di alcuni processi, oppure di alcune storie (con il permesso degli uomini d'affari), disfarsi di processi non essenziali, chiedere aiuto e come ultima possibilità chiedere all'utente di spostare alcune storie per la release successiva.

Appena una storia è completata si verificano anche i test funzionali.

Se le storie poggiano sulla responsabilità collettiva di tutto il gruppo ed è tutto il gruppo che le stima, i processi vengono scelti individualmente prima di essere stimati. Poi ognuno si stima i propri processi.

Esistono inoltre dei processi che non sono direttamente collegati ai bisogni degli utenti, ma che servono all'intero gruppo di sviluppatori per migliorare il proprio lavoro. Per esempio strumenti per velocizzare la costruzione dei test, oppure riorganizzazione delle librerie di oggetti riutilizzabili da chiunque.

Lavori abbastanza complessi che non possono essere compresi nella normale costruzione del software necessario per lo sviluppo del singolo processo. Queste attività possono diventare dei nuovi processi a cui vengono assegnate rispettive priorità e programmati assieme agli altri processi.

#### **4.4. Perché tutto ciò funziona.**

Il tempo speso a pianificare è veramente poco: circa mezza giornata ogni 15 giorni di lavoro. Non è necessario spendere più tempo a causa delle mutabili condizioni.

Già dopo la prima settimana possiamo sapere se stiamo rispettando la tabella di marcia grazie alla persona incaricata di tenere traccia delle stime e del progresso del lavoro. Rimane tutto il tempo per localizzare i problemi e correggere le stime senza essere costretti a chiedere all'utente di fare dei cambiamenti alle specifiche.

Fare in modo che gli sviluppatori scelgano i processi prima di stimarli permette che questi vengano assunti in piena responsabilità.

Accorgersi di riuscire a lavorare solo 8 giorni su 15 stimola a non soffermarci troppo su un solo processo. D'altra parte il lavoro in coppia

assicura che non venga trascurata la qualità finale. Questi due fattori spingono a realizzare semplicemente solo quello di cui abbiamo bisogno per completare il processo.

L'intero processo non deve diventare troppo opprimente a causa dei carichi di lavoro eccessivi. Questo viene evitato mantenendo gli 8 giorni di effettivo lavoro su 15.

Un'ultima osservazione. Se il gruppo di programmatori è molto piccolo, non è assolutamente necessario eseguire la pianificazione iterativa. Si riesce infatti a controllare e ad organizzare proficuamente il progetto anche senza impiegare tempo nella divisione del lavoro, grazie al facile controllo del sistema.