INGEGNERIA DEL SOFTWARE

INTRODUZIONE

Avvertenza: gli appunti si basano sul corso di Ingegneria del Software tenuto dal prof. Picco della facoltà di Ingegneria del Politecnico di Milano (che ringrazio per aver acconsentito alla pubblicazione). Essendo stati integrati da me con appunti presi a lezione, il suddetto docente non ha alcuna responsabilità su eventuali errori, che vi sarei grato mi segnalaste in modo da poterli correggere.

e-mail: webmaster@morpheusweb.it
web: http://www.morpheusweb.it

INTRODUZIONE

ENGINEERING TASKS

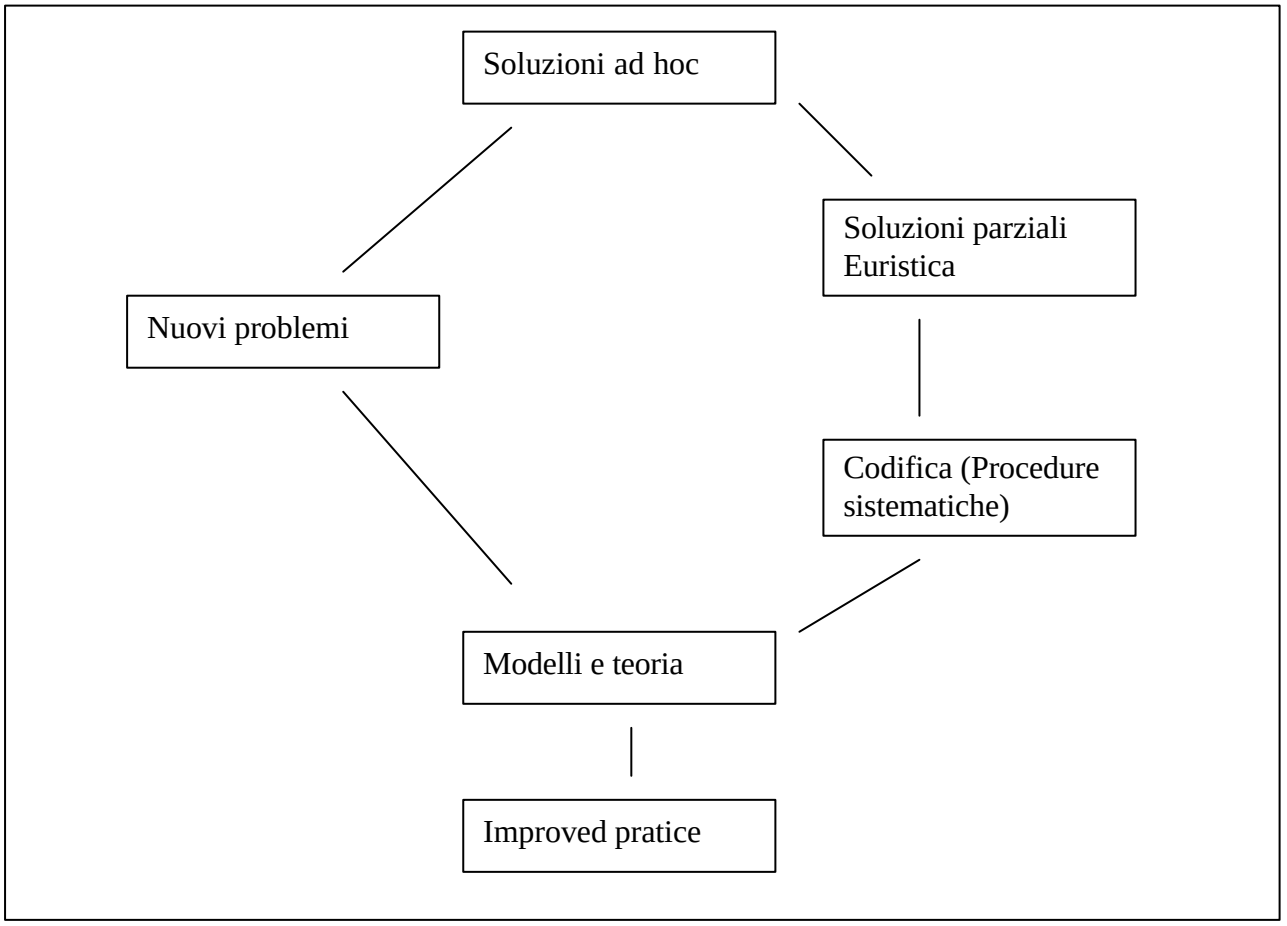
DESIGN DI ROUTINE

- Soluzione a problemi noti
- Riutilizzo di soluzioni precedenti

DESIGN INNOVATIVO

- Soluzioni innovative per problemi nuovi

CICLO DI VITA



INGEGNERIA DEL SOFTWARE VS INGEGNERIA TRADIZIONALE

- Praticata (e pensata) in maniera non sistematica
- Meno stabile ed organizzata di quella tradizionale
- Non esistono standard per la specifica del design del software

DIFFERENZE ED ANALOGIE

I ponti sono generalmente costruiti nei tempi decisi, con un certo budget e non cadono. Non è così per il software.
Perché?

L'ingegneria tradizionale presuppone:

- Dettaglio nel design
- Modelli per validare design alternativi
- Poca flessibilità nelle specifiche
- Precessi standard

Questi punti non sono validi per l'ingegneria del software.

DIFFERENZA 1

Le specifiche dei prodotti software non possono essere congelate perché il modo in cui le applicazioni vengono usate cambia in continuazione

DIFFERENZA 2

Ci sono secoli di conoscenze per l'ingegneria tradizionale, cosa non vera per l'ingegneria del software.

SKILL RICHIESTI ALL'INGEGNERE DEL SOFTWARE

Non basta saper programmare.

Un programmatore:

- sviluppa un programma
- lavora su specifiche conosciute
- lavora da solo

Un ingegnere del software

- identifica i requisiti e sviluppa le specifiche
- realizza componenti che saranno combinati con altri, sviluppati e manutenuti da altri. Tali componenti potranno far parte di diversi sistemi
- lavora in team

Il software implementa una macchina che deve interagire con l'ambiente esterno.
Gli ingegneri del software devono essere capaci di capire ed analizzare gli ambienti esterni.
Gli ambienti esterni sono il posto dove trovare i requisiti.

SVILUPPO STORICO

SITUAZIONE INIZIALE

- Il software è visto come un'arte
- I computer sono usati per fare calcoli (il dominio applicativo è molto ristretto)
    - Problemi matematici
    - I designers sono anche gli utenti
    - Il ciclo di vita non è esteso
- Arte della programmazione
    - Linguaggi di basso livello
    - Poche risorse

Fattori di qualità di un software sono velocità ed utilizzo della memoria (adesso questo vale solo per sistemi embedded o processori dedicati

DALL'ARTE ALL'ARTIGIANATO

- Si passa dai calcoli alla gestione delle informazioni.
- Nuovo software
    - I progettisti non sono più gli utenti
    - Nascono le software house
- Si usano linguaggi di alto livello
- Primi grandi progetti e primi grandi fiaschi (ci si rende conto che l'approccio singolo non è scalabile per grandi progetti)
    - Problemi: tempi e budget, le persone non riescono a cooperare, specifiche errate (l'enfasi è solo sulla programmazione)

REQUISITI PER L'INGEGNERIA DEL SOFTWARE

Per ovviare i problemi visti è necessario un approccio metodico e sistematico basato su conoscenze scientifiche (approccio ingegneristico)

- Metodi e standard di sviluppo
- Pianificazione e gestione
- Automazione
- Qualità (per dei sistemi critici il fallimento può causare rischi o perdite finanziarie ed umane)

Si passa alla fase industriale dell'ingegneria del software

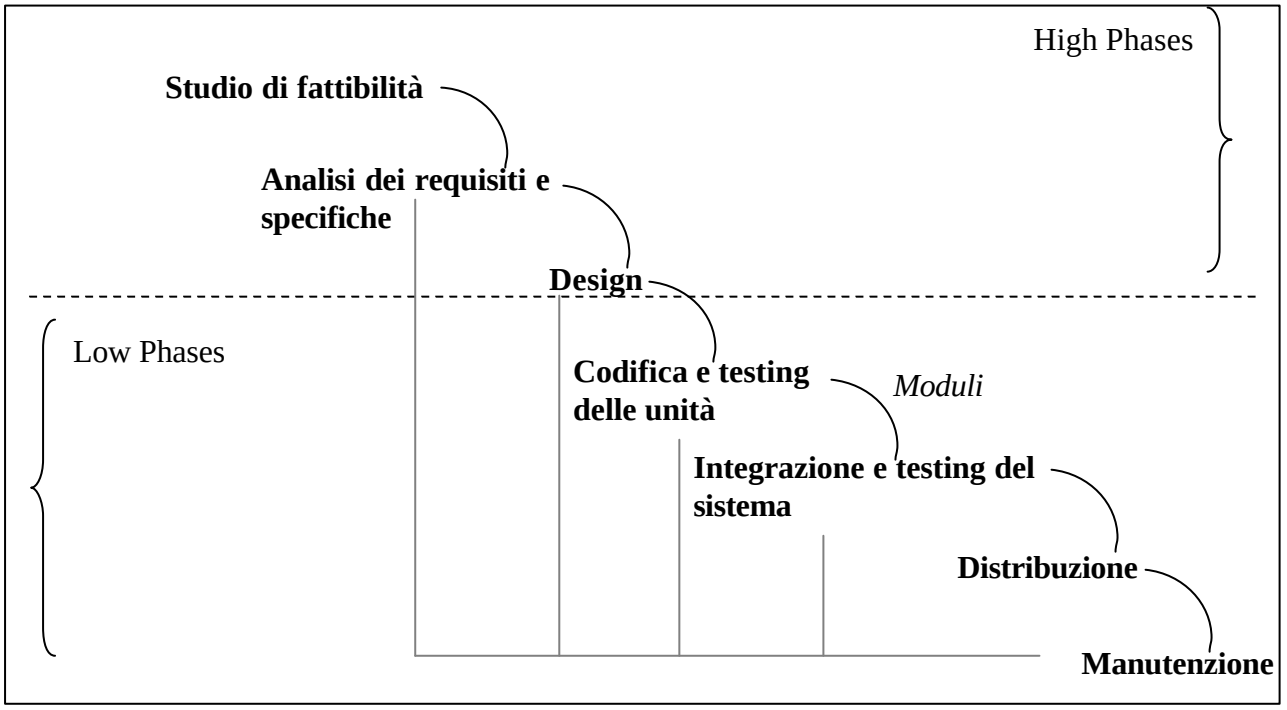
CICLO DI VITA DEL SOFTWARE

Occorre capire qual è il ciclo di vita del software e come gestirlo in maniera corretta.
Il ciclo di vita comprende le fasi di produzione a partire da quando si decide di utilizzare un software e comprende la manutenzione e la dismissione, anche se non tutti i software seguono il ciclo di vita fino in fondo.

MODELLO A CASCATA

Il modello identifica le fasi e le attività da svolgere durante il progetto del software. Trae ispirazione dalle attività ingegneristiche tradizionali.

L'idea è descrivere con chiarezza l'output di ogni fase che è l'input di quella successiva.



FASE 1: STUDIO DI FATTIBILITÀ

- **Analisi costi/benefici**: Valutazione preliminare dei costi e dei benefici di un'applicazione, per stabilire se si debba avviarne lo sviluppo, quali siano le alternative possibili, quali le scelte più ragionevoli, e quali le risorse finanziarie e umane necessarie per l'attuazione del progetto.

- Si sceglie se il prodotto deve essere *realizzato* oppure *comprato*, si valutano risorse alternative

- Si produce un documento con lo **studio di fattibilità** con:

    - descrizione dei problemi (necessità dell'utente)

    - scenario delle soluzioni possibili (si cominciano ad abbozzare alcune soluzioni tecniche)

    - costi per le differenti soluzioni

Qui si decide se andare avanti o meno con lo sviluppo del software.

FASE 2: ANALISI DEI REQUISITI E SPECIFICHE

- Si analizza il **dominio** in cui l'applicazione opererà (ambiente, vincoli di natura tecnologica…)

- Si identificano i **requisiti**: si stabiliscono funzionalità (requisiti funzionali), vincoli e obiettivi consultando gli utenti (tipicamente il committente).
  Un modo possibile di descrivere i requisiti funzionali consiste nel fornire una versione iniziale del Manuale Utente.

- Si derivano le **specifiche** per il software: richiede un'interazione con l'utente ed una comprensione delle proprietà del dominio.

- Viene prodotto il **RASD** (Requirements Analisys and Spcification Document) che deve essere *compatto conciso* e *consistente*

Alcune di queste cose sono già state fatte nella fase 1, ma qui occorre arrivare al massimo livello di dettaglio.

Nella fase di analisi dei requisiti e delle specifiche ci si deve concentrare sui seguenti punti (**le 5 w**)

- **Who** (chi userà il sistema)

- **Why** (perché deve essere sviluppato e perché gli utenti dovrebbero usarlo)

- **What** vs HOW (cosa apporterà il sistema, e non su come deve essere fornita)

- **Where** (dove sarà usato, su quale architettura)

- **When** (quando e per quanto tempo sarà usato)

Il **RASD**: dovrà essere: **preciso**, **completo** e **consistente** e potrà includere un manuale utente preliminare.

In questa fase viene anche definito *il Piano di Test di Sistema* (**PTS**) che descrive le modalità con cui, al termine dello sviluppo, nella fase di integrazione, si possa verificare il sistema sviluppato rispetto ai requisiti fissati

Anche questo documento andrebbe sottoscritto dal committente

FASE 3: DESIGN (PROGETTAZIONE)

Si definisce l'architettura generale (hardware e software) del sistema

- moduli
- relazioni
- interazioni (da cui capisco il comportamento a run-time del sistema)

Si descrivono le funzioni che il sistema deve svolgere, ciascuna delle quali verrà trasformata in uno o più programmi eseguibili; l'*obiettivo* è scomporre il problema in sottoproblemi in modo da ridurre la complessità

L'architettura software può essere composta da moduli, evidenziando quali siano le funzionalità offerte dai diversi moduli e le relazioni tra i moduli

Il risultato dell'attività di progettazione è il *Documento* di *Specifiche* di *Progetto* (**DSP**) nel quale la definizione dell'architettura software può anche essere data in maniera rigorosa, o addirittura formale, usando opportuni linguaggi di specifica di progetto

FASE 4: CODIFICA E TESTING DELLE UNITÀ

Il progetto viene realizzato come insieme di programmi o unità di programmi (moduli).

- Ogni modulo è implementato usando il linguaggio di programmazione scelto
- Ogni modulo è testato singolarmente dallo sviluppatore (in isolamento)

Il testing delle unità serve per verificare che ciascuna soddisfi le specifiche richieste.

Il prodotto è:

- codice sorgente
- risultati dei test (dati di ingresso ed output)
- documentazione (commenti, scelte tecniche…)

FASE 5: INTEGRAZIONE E TESTING DEL SISTEMA

I moduli sono integrati in (sotto)sistemi e questi vengono testati, c'è poi un test del sistema completo.

Questa fase e la precedente possono essere integrate in uno schema di implementazione incrementale.

Alfa e Beta test

- **Alfa test** quando il sistema è rilasciato per l'uso, ma all'interno dell'organizzazione del produttore
- **Beta test** quando si ha un rilascio controllato a pochi e selezionati utenti del prodotto

FASE 6: DISTRIBUZIONE

Lo scopo è distribuire l'applicazione e gestire le diverse installazioni e configurazioni tra i vari clienti.
Prima era fatta on site, adesso può anche essere fatta in remoto.

FASE 6: MANUTENZIONE

È la fase più lunga del ciclo di vita di un prodotto software (oltre il 50% dei costi complessivi del ciclo di vita)

Circa l'80% del budget IT è speso in manutenzione.

La manutenzione comporta la correzione degli errori che non erano stati scoperti nelle fasi precedenti, migliorando la realizzazione delle unità del sistema ed aumentando i servizi forniti man mano che si richiedono nuovi requisiti.

PRECISAZIONI

EVOLUZIONE

Perché un software dovrebbe evolvere? I motivi possono essere molteplici.

- Cambiamenti del contesto (esempio € vs £)
- Cambiamento nei requisiti
- Specifiche sbagliate
- Requisiti sconosciuti in anticipo

TIPOLOGIE DI CAMBIAMENTI:

- Correttivi (software bacato) ~ 20%
- Adattativi (ci si deve adattare a cambiamenti nel sistema o nell'ambiente) ~ 20%
- Migliorativi (per avere delle funzionalità in più) ~ 50%

BUONE ABITUDINI

Prima modificare il progetto, poi l'implementazione applicando le modifiche in modo consistente in tutti i documenti.

COME FRONTEGGIARE L'EVOLUZIONE

L'obiettivo è anticipare i cambiamenti. Il software deve essere generato per rimanere aperto, deve essere possibile cambiarlo in modo semplice dopo che è stato sviluppato. I cambiamenti devono essere facilmente realizzabili ed a basso costo.

Non bisogna confondere **evoluzione** con **correzione**.

Spesso il software non è stato progettato per essere modificato facilmente Si apportano modifiche intervenendo direttamente sui programmi, senza modificare, se è il caso, la documentazione di progetto e di test, la specifica dei requisiti, etc.

Un problema molto sentito dai produttori di software è quello di ***"recuperare"*** le applicazioni esistenti.

La re-ingegnerizzazione del software (reverse engineering) consiste nella possibilità di riportare software ormai poco strutturato e non documentato in uno stato in cui sia possibile poi ripartire in modo sistematico nella manutenzione

DATI SUGLI ERRORI

- Ispezioni sistematiche possono trovare fino al 50-75% degli errori
- Moduli con un controllo di flusso complesso può facilmente contenere più errori.
- Spesso i test coprono soltanto il 50% del codice
- Il codice distribuito contiene il 10% degli errori trovati nel testing
- Gli errori iniziali sono scoperti tardi ed il costo per eliminarli aumenta con il passare del tempo
- Eliminare errori da un grosso sistema costa più (4-10 volte) che in un sistema piccolo

- L’eliminazione di errori introduce nuovi errori
- I grossi sistemi tendono a stabilizzarsi ad un certo livello di imperfezione

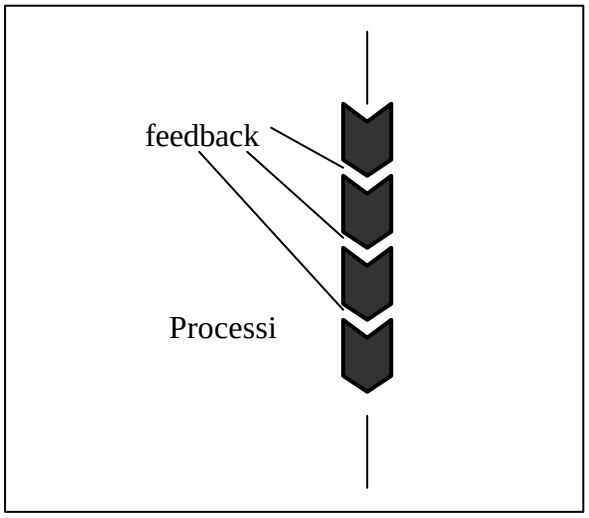
VARIANTI DEL MODELLO A CASCATA

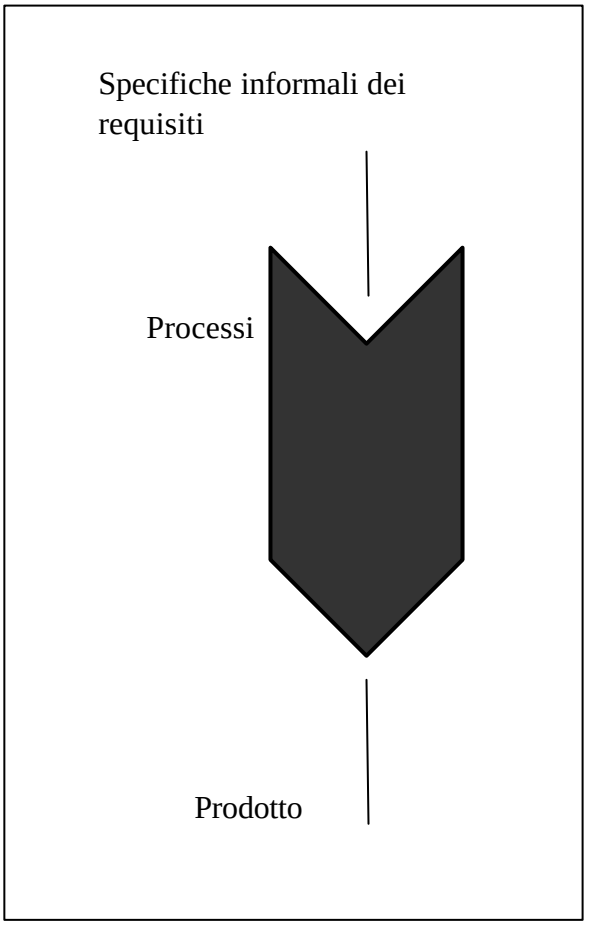
A volte non servono tutte le fasi:

- software per uso personale
- utente che appartiene alla stessa organizzazione
- applicativi per la vendita sul mercato

Il ciclo a cascata può essere deleterio. Bisogna essere preparati a più cicli di iterazione. Il riciclo non può essere eliminato.

La cascata può essere vista come una **black box**, in cui inseriamo i requisiti e tiriamo fuori il prodotto.
In realtà si vuole un monitoraggio all’interno del processo.
Non voglio solo che il prodotto rispetti le specifiche ma voglio anche un riscontro nelle varie fasi.
Si vuole un feedback per supportare la **flessibilità**.





VERIFICA E VALIDAZIONE

- Verifica: stiamo producendo il prodotto nel modo corretto?

- Validazione: stiamo producendo il prodotto giusto? è una verifica rispetto agli input

PROCESSI FLESSIBILI

Il modello a cascata non va sempre bene, esistono altri modelli più flessibili che servono a tenere conto fin dall'inizio dei cambiamenti.
L'idea è quella di avere dei processi incrementali in cui ho un feedback ad ogni passo.

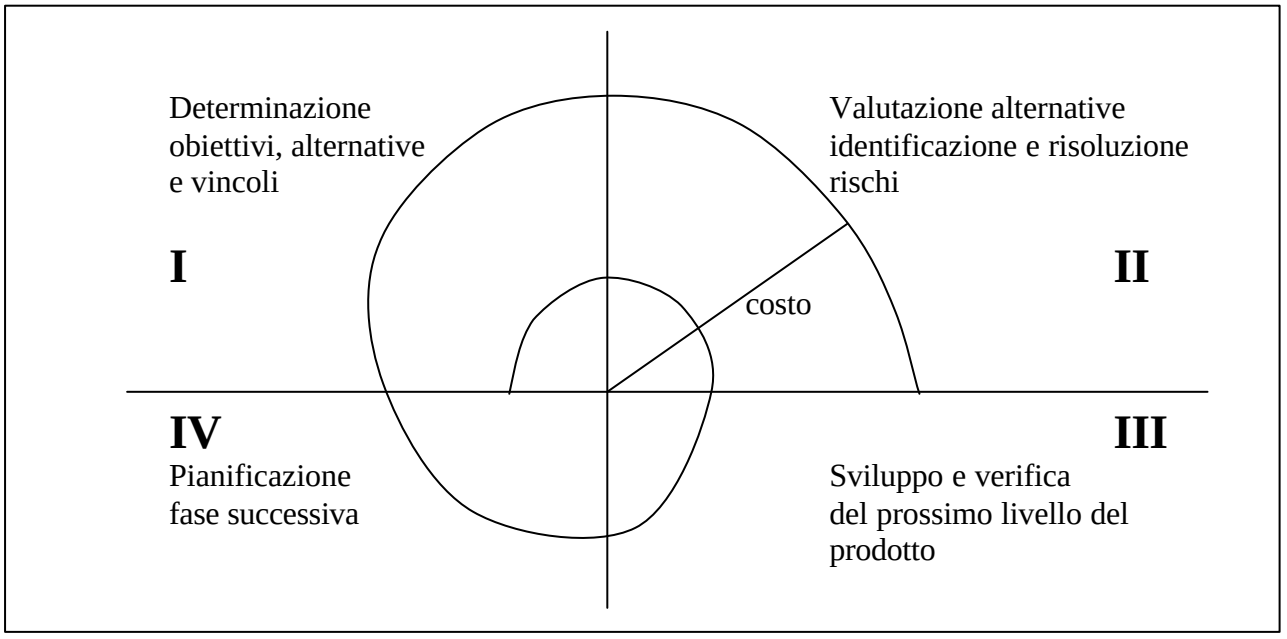
PROTOTIPAZIONE

Salta alcune fasi. Si va all'implementazione (prototipo) che non ha tutte le funzionalità ma un sottoinsieme di funzioni ritenute rilevanti.
Ci sono prototipi **throw-away,** che vengono buttati via oppure **evolutivi**, che consentono di scrivere il software in maniera incrementale.
Si parla di **incremental delivery** quando si sottopone il prototipo all'approvazione del committente ad ogni passo.

MODELLO A SPIRALE

Proposto per supportare l' analisi dei rischi

Modello di tipo ciclico, dove il raggio della spirale rappresenta il costo accumulato durante lo svolgimento del progetto



Ogni ciclo della spirale rappresenta una fase:
il più interno può essere lo studio di fattibilità, il successivo la definizione dei requisiti, il successivo ancora la progettazione, etc.
Non sono definite fasi a priori
Man mano che si cicla nella spirale, i costi aumentano.

Ogni ciclo della spirale passa attraverso i quadranti del piano che rappresentano i seguenti passi logici:

I. determinazione di obiettivi, alternative e vincoli

II. valutazione di alternative, identificazione e risoluzione di rischi
ad esempio sviluppo di un prototipo per validare i requisiti

III. sviluppo e verifica del prossimo livello del prodotto

modello evolutivo, se i rischi riguardanti l'interfaccia utente sono dominanti; a cascata se è l'integrabilità del sistema il rischio maggiore; trasforma-zionale, se la sicurezza è più importante

IV. pianificazione della fase successiva
si decide se continuare con un altro ciclo della spirale

Costituisce un meta-modello dei processi software, cioè un modello per descrivere modelli

Non è detto che per un ciclo della spirale si adotti un solo modello di sviluppo

Può descrivere uno sviluppo incrementale, in cui ogni incremento corrisponde a un ciclo di spirale

Può descrivere il modello a cascata (quadranti I e II corrispondenti alla fase di studio di fattibilità e alla pianificazione del progetto; quadrante III corrisponde al ciclo produttivo)

Differisce dagli altri modelli perché considera esplicitamente il fattore rischio

Boehm suggerisce di considerare, per ciascun ciclo, le seguenti voci:

- Obiettivi
- Vincoli
- Alternative
- Rischi
- Soluzioni per i rischi
- Risultati
- Piani
- Decisioni

EXTREME PROGRAMMING

Agli inizi dell'informatica, il software veniva sviluppato in modo caotico e disorganizzato, senza un progetto chiaro su cui basarsi. Questo metodo funziona bene soltanto per software di piccole dimensioni, ma al crescere del sistema diventa molto difficile da gestire. Apportare modifiche o aggiungere funzionalità al software diventa quasi impossibile. Si perde la maggior parte del tempo a correggere bachi, piuttosto che ha sviluppare il sistema.

Per risolvere questi problemi sono state proposte molte metodologie software. Una metodologia impone agli sviluppatori un preciso processo di sviluppo nel tentativo di renderlo più prevedibile e più efficiente. Queste metodologie sono state soprannominate "metodologie pesanti" (Heavyweight o Monumental Methodologies) a causa del fatto che predicavano la produzione di una miriade di documenti durante tutto il processo di sviluppo.

Negli ultimi anni ne sono state proposte di nuove, che rappresentano validi compromessi tra il non seguire un processo di sviluppo e seguirlo troppo strettamente. Queste metodologie sono soprannominate "metodologie leggere" (Lightweight o Agile Methodologies) e sono caratterizzate da due aspetti principali:

- I metodi leggeri sono adattivi più che predittivi. I metodi pesanti cercano di programmare lo sviluppo nel dettaglio e in modo da soddisfare tutte le specifiche. Nel caso di cambiamenti di specifica, c'è il rischio che si incrini l'intera struttura. Le nuove metodologie invece progettano programmi pensati per cambiare nel tempo.

- I metodi leggeri sono people-oriented invece che process-oriented. L'approccio prevede di adattare il processo di sviluppo alla natura dell'uomo e non costringere l'uomo ad adattarsi in modo da far si che diventi un'attività piacevole per gli sviluppatori.

Una forte spinta allo sviluppo e alla diffusione di queste nuove metodologie è stata data dal successo del software "open source", che viene sviluppato in modo decentrato e anarchico e sembra sfidare le regole del project management.

Extreme Programming

Tra tutte le metodologie leggere, l'XP è quella che ha ottenuto e sta tuttora ottenendo la maggiore attenzione. XP affonda le sue radici nella comunità Smalltalk e in particolare nella stretta collaborazione tra Kent Back e Ward Cunningham verso la fine degli anni '80.

Nessuna delle tecniche proposte dall'XP è nuova: la novità sta nell'averle riunite tutte insieme.

XP si basa su **quattro valori fondamentali**:

- **Comunicazione**: ci deve essere grande comunicazione sia tra i programmatori che tra programmatori e clienti.

- **Feedback (testing):** il collaudo è una delle questioni fondamentali. E' necessario avere più codice per il test che per il programma vero e proprio. Ogni programmatore deve scrivere il programma di test parallelamente se non prima di scrivere il codice effettivo.

- **Semplicità**: il codice deve essere il più semplice possibile. Complicare il codice tentando di prevedere futuri riutilizzi è controproducente sia come qualità di codice prodotto sia come tempo necessario. D'altra parte il codice semplice e comprensibile è il più riutilizzabile.

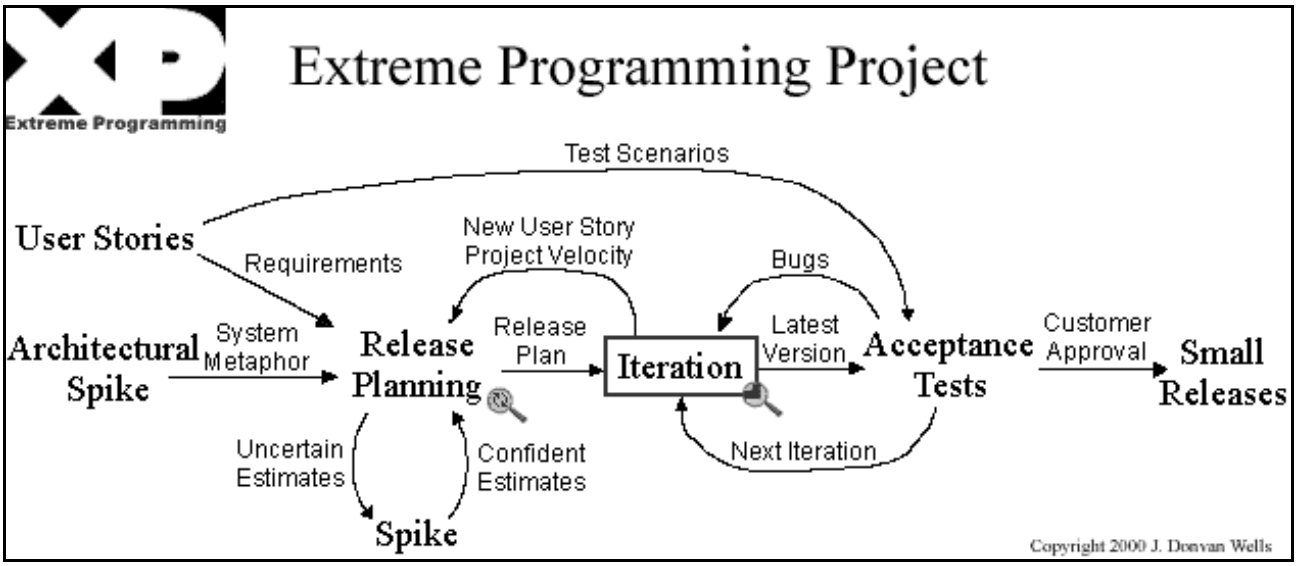
- **Coraggio**: non si deve avere paura di modificare il sistema, ma deve essere ristrutturato in continuazione, ogni volta che si intravede un possibile miglioramento.
  A partire da questi principi XP propone una dozzina di tecniche che gli sviluppatori dovrebbero utilizzare. La maggior parte di queste è vecchia, già provata e testata, ma molto spesso dimenticata da molti. Il merito di XP sta nell'averle riunite in un insieme sinergico, dove ognuna rinforza le altre.

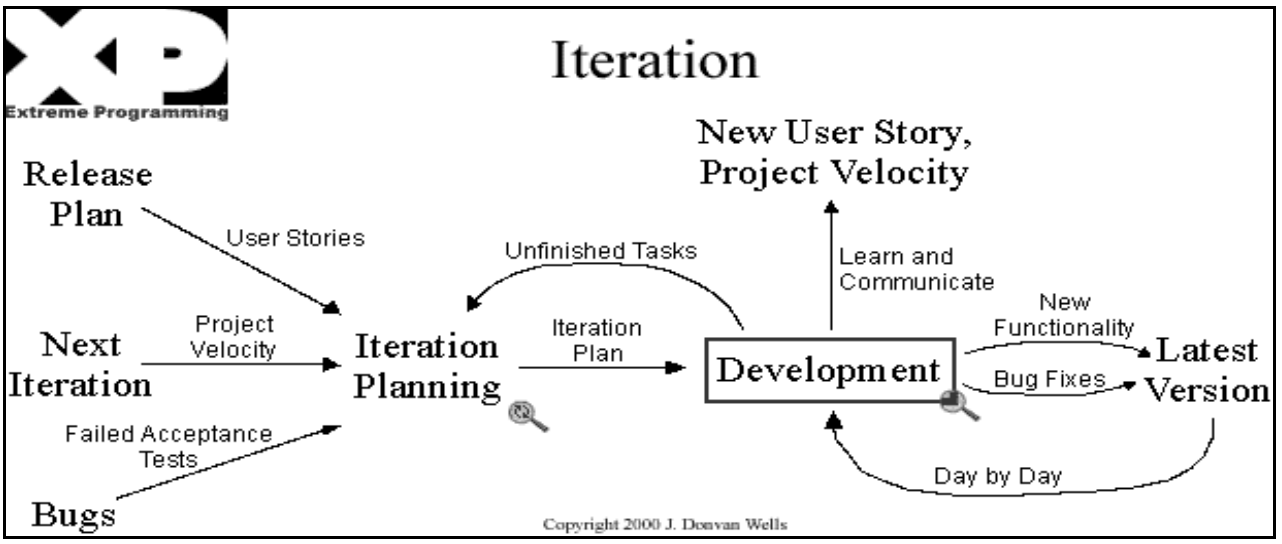
Il processo di sviluppo di XP è evoluzionale e iterativo e si basa sul refactoring ad ogni iterazione di un semplice sistema base. Tutto il design è incentrato sull'iterazione attuale e non ci si concentra sulle necessità future. Il risultato è un processo di design che combina disciplina e adattività in un modo che rende questa metodologia la migliore tra quelle adattive.

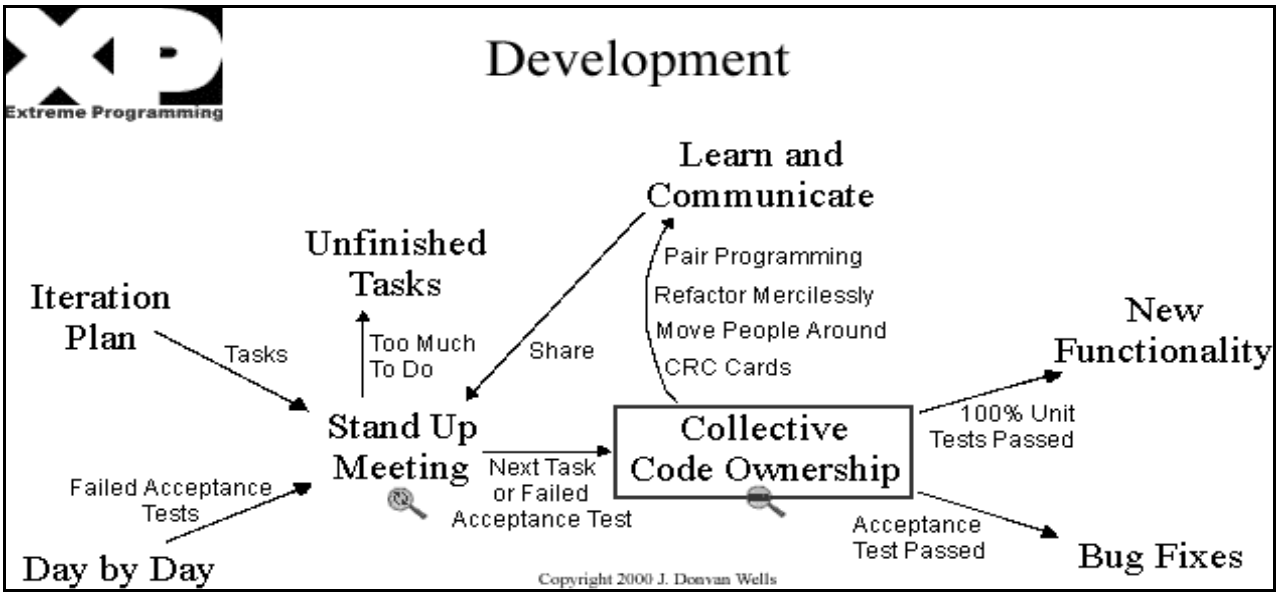
THE RULES AND PRACTICES OF EXTREME PROGRAMMING

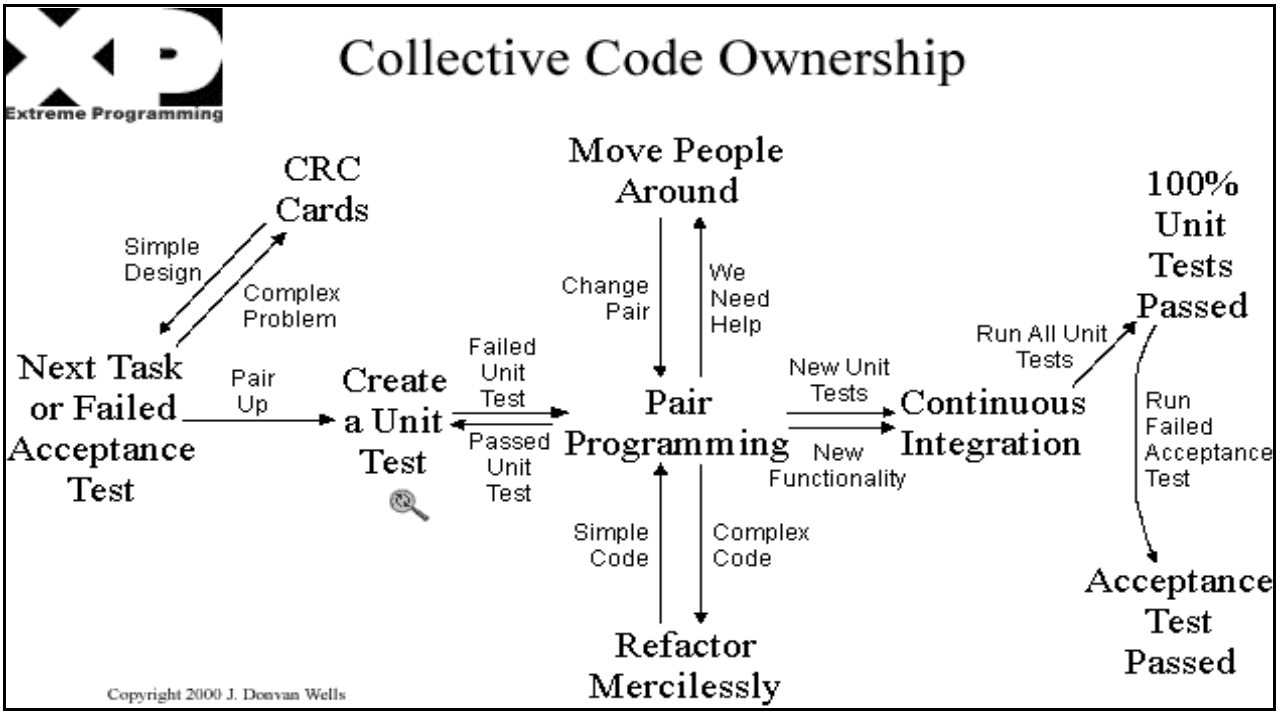
Le regole e le tecniche che XP propone si possono suddividere in quattro categorie: regole sul planning, sul design, sulla codifica e sul testing.

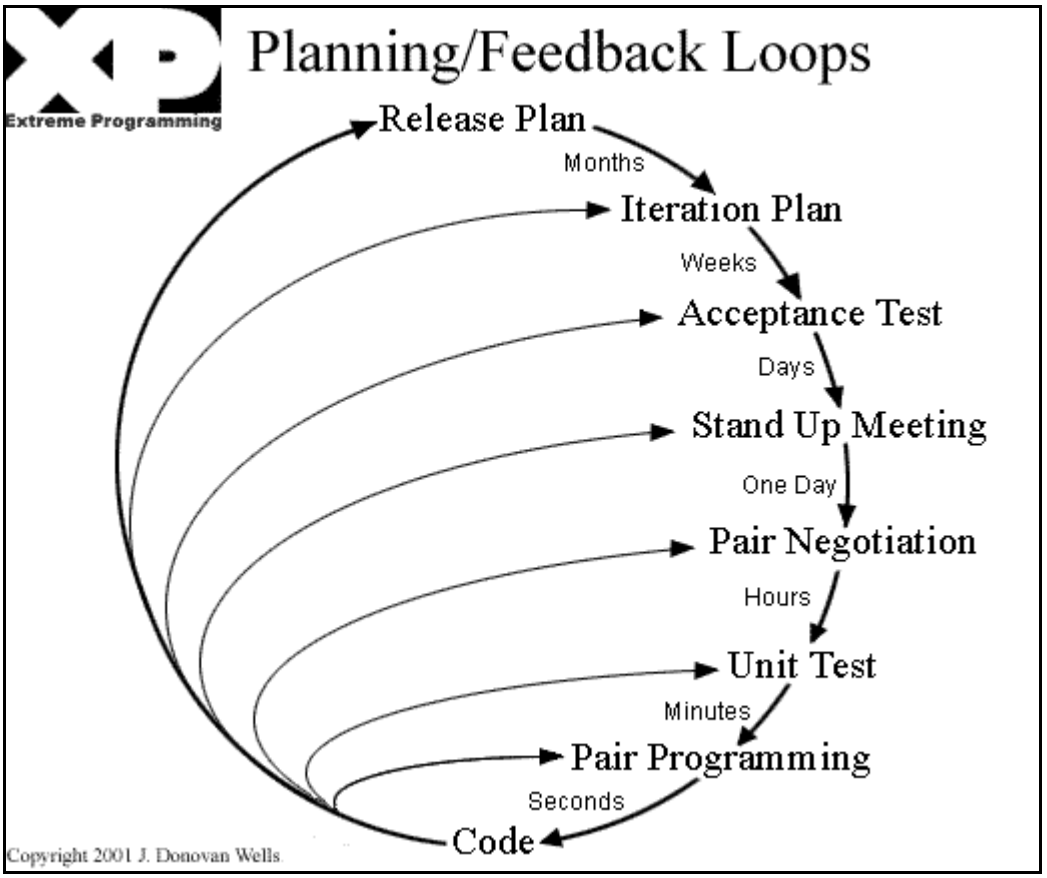
Di seguito è riportato il diagramma principale di un progetto sviluppato utilizzando XP. Il significato della terminologia viene chiarito nelle varie regole.




Extreme Programming
Iteration
New User Story,
Project Velocity
Release
Plan
User Stories
Unfinished Tasks
Learn and
Communicate
New
Functionality
Next
Iteration
Project
Velocity
Iteration
Planning
Iteration
Plan
Development
Latest
Version
Bug Fixes
Failed Acceptance
Tests
Day by Day
Bugs
Copyright 2000 J. Donvan Wells


Extreme Programming
Development
Learn and
Communicate
Unfinished
Tasks
Pair Programming
Refactor Mercilessly
Move People Around
CRC Cards
Iteration
Plan
Tasks
Too Much
To Do
Share
New
Functionality
Stand Up
Meeting
Next Task
or Failed
Acceptance Test
Collective
Code Ownership
100% Unit
Tests Passed
Failed Acceptance
Tests
Acceptance
Test Passed
Bug Fixes
Day by Day
Copyright 2000 J. Donvan Wells


Extreme Programming
Collective Code Ownership
CRC
Cards
Move People
Around
100%
Unit
Tests
Passed
Simple
Design
Complex
Problem
Change
Pair
We
Need
Help
Run All Unit
Tests
Next Task
or Failed
Acceptance
Test
Pair
Up
Create
a Unit
Test
Failed
Unit
Test
Passed
Unit
Test
Pair
Programming
New Unit
Tests
New
Functionality
Continuous
Integration
Run
Failed
Acceptance
Test
Simple
Code
Complex
Code
Acceptance
Test
Passed
Refactor
Mercilessly
Copyright 2000 J. Donvan Wells



IL PLANNING

- User stories are written.
- Release planning creates the schedule.
- Make frequent small releases.
- The Project Velocity is measured.
- The project is divided into iterations.
- Iteration planning starts each iteration.
- Move people around.
- A stand-up meeting starts each day.
- Fix XP when it breaks.

USER STORIES ARE WRITTEN.

Il primo passo nella pianificazione del lavoro consiste nello scrivere le cosiddette "storie" (User Stories). Le storie sono documenti scritti dal cliente che descrivono le funzionalità che il software deve avere. Sono simili agli scenari d'uso ma non sono limitate a descrivere interfacce utente. Sono costituite da circa tre frasi scritte dal cliente nel linguaggio del cliente.

A cosa servono?

Vengono utilizzate nei release-planning meeting per determinare i tempi di rilascio.
Sono utilizzate come sostituti del documento dei requisiti.
Servono per la creazione dei test di accettazione (Sono uno o più test automatizzati che devono essere realizzati in modo da poter verificare che una storia sia stata implementata correttamente).
La storie sono molto diverse dalle tradizionali specifiche dei requisiti. In particolare sono diverse per quanto riguarda il livello di dettaglio: la storie devono soltanto contenere abbastanza dettagli per poter realizzare una stima a basso rischio (low-risk estimate) di quanto tempo sia necessario per implementarla. Quando occorrerà effettivamente realizzarla, gli sviluppatori dovranno rivolgersi nuovamente al cliente per farsi fornire una descrizione dettagliata dei requisiti.

Una volta scritte le storie, gli sviluppatori stimano i tempi di realizzazione per ognuna di esse. I tempi devono essere compresi tra 1 e 3 settimane. Se si superano le 3 settimane significa che la storia è troppo complessa e va suddivisa, se si va sotto la singola settimana significa che si è arrivati ad un livello di dettaglio troppo basso e bisogna unirla con un'altra. Dalle 60 alle 100 storie costituiscono un numero adeguato per creare un release plan.

RELEASE PLANNING CREATES THE SCHEDULE.

Il meeting di release (release planning meeting) è una riunione in cui si crea il piano di rilascio (release plan) del progetto. Questo verrà poi utilizzato per realizzare i piani di iterazione (iteration plans) di ogni singola iterazione.

Per il team di sviluppo la sostanza di questo meeting consiste nello stimare i tempi di sviluppo di ogni storia. Nella stima si ragione in base a settimane di programmazione ideal (ideal programming weeks), cioè si stima il tempo necessario se non ci fosse altro da fare a parte quella storia. Tuttavia si tiene conto del tempo necessario per il testing. Dopodiché il cliente decide quali storie far sviluppare prima in base alle proprie necessità e al tempo che esse richiedono.

Le storie sono scritte su delle carte (simili alle CRC). Sviluppatori e clienti muovono le carte su un grande tavolo fino ad ottenere un set di storie da sviluppare alla prima (o alla prossima iterazione). Si cerca in ogni caso di ottenere il più presto possibile un sistema usabile e testabile dal cliente anche se con poche funzionalità.

Alla fine si ottiene un release plan che specifica esattamente quali storie vanno implementate ad ogni release del sistema e le date di rilascio.

MAKE FREQUENT SMALL RELEASES.

Il team di sviluppo deve rilasciare frequentemente nuove versioni al cliente molto spesso. Questo è importante per ottenere del feedback dal cliente il più presto possibile. Ciò permette di accorgersi immediatamente di eventuali errori di design.

PROJECT VELOCITY.

La project velocity è una misura della velocità di sviluppo del progetto. Per calcolare la velocità di progetto basta sommare le stime temporali delle storie che sono state sviluppate nell'ultima iterazione. Questa potrà poi essere utilizzata nel release planning meeting per quantificare quante storie si riusciranno a sviluppare nella prossima iterazione.

E' normale che la velocità di progetto cambi leggermente, ma se cambia drasticamente per più di una iterazione sarà necessario un nuovo release planning meeting per ristimare e rinegoziare i tempi di release.

THE PROJECT IS DIVIDED INTO ITERATIONS (ITERATIVE DEVELOPMENT).

Lo sviluppo basato su iterazioni dona agilità al processo di sviluppo. Il processo di sviluppo va diviso in periodi che vanno da 1 a 3 settimane e in ciascuno di questi si elabora una nuova iterazione. I periodi vanno mantenuti costanti nel tempo, sono come il battito del cuore del progetto, danno il ritmo. Ciò rende possibile misurare i progressi e pianificare in modo semplice.

Non bisogna pianificare in anticipo l'implementazione di funzionalità che non devono essere sviluppate nell'iterazione corrente; bisogna invece organizzare un iteration planning meeting all'inizio di ogni iterazione per pianificare ciò che dovrà essere fatto. Inoltre non bisogna mai guardare oltre per tentare di implementare funzionalità non pianificate per l'iterazione attuale.

Le scadenze di ogni iterazioni vanno prese sul serio. Bisogna tener conto del proprio progresso durante un iterazione; se ci si trova in difficoltà e si pensa di non riuscire a finire in tempo occorre organizzare un nuovo iteration planning meeting.

A prima vista può sembrare sciocco organizzare iterazione anche di una sola settimana, ma alla fine conviene. Pianificando ogni iterazione come se fosse l'ultima vi aiuterà a rispettare i tempi.

ITERATION PLANNING STARTS EACH ITERATION.

All'inizio di ogni iterazione si organizza un iteration planning meeting per pianificare come sviluppare le storie dell'iterazione attuale, organizzando il lavoro in programming task. I programming task vengono scritti su delle carte (task cards) che costituiranno il piano dettagliato per l'iterazione. I programmatori scelgono dei task da sviluppare e fanno una stima temporale sul tempo di cui avranno bisogno per completarli. Ogni task dovrebbe avere una durata che va da 1 a 3 giornate ideali. Come per le storie, task troppo corti vanno uniti tra loro e task troppo lunghi vanno suddivisi.

E' importante valutare se si sono assegnate troppe storie all'iterazione attuale. Per fare questo si utilizza la project velocity: si sommano i tempi necessari per ogni programming task e si verifica che non superino la project velocity dell'ultima iterazione. Se ci sono troppi task bisogna accordarsi con il cliente per decidere quali storie posticipare fino alla prossima iterazione (Snow plowing). Se invece avanza del tempo, si può anticipare lo sviluppo di una o più storie.

MOVE PEOPLE AROUND

Le persone che lavorano ad un progetto, dovrebbero spostarsi da un area all'altra in modo da ottenere una conoscenza il più possibile generale sullo stesso. E' sconsigliabile che sia solo una persona a lavorare in un area, in quanto se questa persona dovesse essere licenziata o dovesse

andarsene, ci si ritroverebbe senza nessuna conoscenza sul codice da lui sviluppato. Se ogni persona inoltre conosce abbastanza bene tutto il sistema, il team sarà molto più flessibile: si potranno spostare liberamente persone da un gruppo ad un altro in modo da equilibrare il carico di lavoro.

DAILY STAND-UP MEETING

Ogni mattina viene organizzato uno stand-up meeting (cioè un meeting in piedi) che coinvolge tutto il team di sviluppo. Si svolge in piedi in modo da evitare che duri più del necessario.

FIX IT WHEN IT BREAKS.

Aggiusta il processo quando non va bene. Le regole di XP andrebbero seguite tutte, ma se ritenete che qualcosa non funzioni non esitate a cambiarla. Ciò non significa che ognuno può fare ciò che vuole, ma che le regole vanno seguite finché tutto il team non decide che una determinata regola è controproducente per il progetto in corso.

IL DESIGN

- Simplicity is the key.
- Choose a system metaphor.
- Use CRC cards for design sessions.
- Create spike solutions to reduce risk.
- No functionality is added early.
- Refactor whenever and wherever possible.

SIMPLICITY IS THE KEY

Un design semplice è molto più veloce e facile da sviluppare di uno complesso e quindi più conveniente. Se trovate qualche soluzione complessa, cercate di modificarla subito in qualcosa di semplice. Inoltre essendo il processo XP un processo iterativo, in futuro sarà più semplice apportare modifiche e miglioramenti..

CHOOSE A SYSTEM METAPHOR

Scegliete una metafora per il sistema in modo cheche i nomi di classi e metodi siano consistenti in tutto il progetto. I nomi sono particolarmente importanti per comprendere il design complessivo del sistema e per facilitare il riutilizzo di codice. Essere in grado di indovinare come si potrebbe chiamare un oggetto quando lo si cerca può far risparmiare un sacco di tempo.

USE CRC CARDS FOR DESIGN SESSIONS.

I membri del team usano le Class-Responsibilities-Collaboration (CRC) per fare il design del sistema tutti insieme. Lavorare con le CRC permette a tutti i membri di contribuire al design del sistema.
Create spike solutions to reduce risk.

Una spike solution è un programma molto semplice che viene realizzato per esplorare le potenziali soluzioni ad un determinato problema o le varie modalità in cui è implementabile una storia. Devono riguardare un singolo problema e non si devono interessare del resto. La maggior parte delle spike non è abbastanza completa per essere integrata nel sistema, quindi andrà gettata e sostituita. Lo scopo della creazione delle spike è di ridurre i rischi legati a problemi tecnici oppure alla stima dei tempi di realizzazione di una storia.

NO FUNCTIONALITY IS ADDED EARLY.

Non va aggiunta nessuna funzionalità extra al sistema che si immagina verrà usata successivamente. Solitamente soltanto il 10% di queste verrà utilizzato effettivamente quindi si stà buttando via il 90% del tempo. Tutti siamo tentati dall'aggiungere funzionalità per migliorare il sistema, ma bisogna trattenersi, ragionando sul fatto che molto probabilmente non verranno utilizzate e che quindi è conveniente concentrare i nostri sforzi sulle richieste attuali.

REFACTOR WHENEVER AND WHEREVER POSSIBLE.

Molti programmatori continuano ad utilizzare design anche quando oramai sono obsoleti e difficilmente mantenibili, per paura di romperli nel modificarli. Ma è veramente conveniente? Secondo la filosofia XP no. XP sostiene che un continuo refactoring del software ne aumenta sensibilmente la qualità e fa risparmiare tempo nel corso dell'intero ciclo di vita del progetto. Il refactoring andrebbe eseguito senza pietà in modo da mantenere il design snello e semplice, rimuovendo funzionalità non più necessarie o duplicate.

CODING

- The customer is always available.
- Code must be written to agreed standards.
- Code the unit test first.
- All production code is pair programmed.
- Only one pair integrates code at a time.
- Integrate often.
- Use collective code ownership.
- Leave optimization till last.
- No overtime.

THE CUSTOMER IS ALWAYS AVAILABLE

Il cliente del progetto deve essere sempre disponibile, non solo per aiutare il team ma perché ne fa parte. Durante tutte le fasi di sviluppo del progetto è necessaria l'interazione con il cliente, preferibilmente faccia a faccia.

Il cliente scrive le user stories e si accorda sulle priorità di sviluppo di queste con i programmatori. Inoltre deve assicurarsi che la maggior parte delle funzionalità del sistema siano descritte nelle user stories.

Dato che le storie sono descrizioni molto astratte delle funzionalità del sistema, gli sviluppatori, per completare il programming task relativo, avranno la necessità di parlare con il cliente per avere maggiori dettagli. Progetti di dimensioni significanti richiederanno un impegno full-time del cliente.

Può sembrare che si richieda un sacco di tempo al cliente, ma ripensandoci gliene si risparmia molto evitandogli di produrre specifiche dettagliate all'inizio consegnando poi un prodotto effettivamente funzionante.

CODE MUST BE WRITTEN TO AGREED STANDARDS.

Il codice deve essere scritto in modo standard, cioè seguendo una impostazione comune a tutto il team. Ciò permette a tutti di leggere facilmente e modificare il codice di altri.

CODE THE UNIT TEST FIRST.

Se si scrive il test prima del codice, successivamente creare il codice sarà molto più semplice. Inoltre si avrà subito modo di verificare che il codice sia funzionante e sia completo in tutte le sue parti.

Creare il programma di test aiuta lo sviluppatore a capire cio che bisogna effettivamente fare e quali siano le responsabilità del codice da scrivere. I requisiti vengono definiti molto bene dal programma di test.

Si può inoltre ragionare in modo iterativo: prima si crea un test per un singolo aspetto del problema in esame. Poi si crea il codice più semplice che passi il test. Successivamente si crea un nuovo test e si aggiunge il codice relativo. Si procede così finche non si è risolto il problema. In questo modo il codice creato sarà semplice e conciso e implementerà soltanto le funzionalità di cui avrete effettivamente bisogno.

ALL PRODUCTION CODE IS PAIR PROGRAMMED.

Tutto il codice definitivo deve essere scritto a quattro mani, da due persone che lavorano sullo stesso computer. Programmare a coppie (pair programming) aumenta la qualità del software senza incrementare i tempi di sviluppo. Inizialmente può sembrare un controsenso ma il codice prodotto da due persone su uno stesso pc è lo stesso di due persone che lavorano separatamente, ma di qualità maggiore. Durante la scrittura del codice, il programmatore che scrive si concentra sui singoli statement mentre l'altro controlla la correttezza di quanto viene scritto e si concentra sugli aspetti architetturali. Abituarsi al pair-programming richiede del tempo ma i risultati sono assicurati.

ONLY ONE PAIR INTEGRATES CODE AT A TIME.

Spesso nello sviluppo di software nascono problemi durante l'integrazione delle varie unità. La maggior parte dei problemi nascono dal fatto che si cerca di integrare al sistema, in parallelo, le nuove unità . Ogni gruppo di programmatori aggiunge la sua unità pensando che vada tutto bene ma nel frattempo un altro gruppo ha aggiunto la propria e le due sono in conflitto tra loro. XP propone di far integrare a ciascuno le proprie parti di codice ma di farlo a turni, cioè in modo sequenziale. Soltanto una coppia alla volta ha quindi il permesso di integrare il proprio codice all'interno del sistema. In questo modo si evitano un sacco di problemi e si riesce a tenere traccia delle versioni del software.

Solitamente per decidere di chi sia il turno si utilizza un token che viene passato da coppia a coppia. Integrando frequentemente il proprio codice riduce il tempo di token e riduce quindi il tempo che bisogna attendere per il proprio turno.

INTEGRATE OFTEN

Gli sviluppatori dovrebbero integrare il proprio codice dopo poche ore e in ogni caso non devono aspettare più di un giorno. Ciò favorisce la comunicazione tra i vari membri del team e permette a tutti di lavorare con l'ultima versione del sistema.

La continua integrazione inoltre permette di evitare o di individuare subito problemi di compatibilità tra le varie unità.

USE COLLECTIVE CODE OWNERSHIP.

Il codice prodotto da una coppia è proprietà di tutto il team. Questo incoraggia tutti i programmatori a contribuire con le proprie idee a tutte le varie aree del progetto. Tutti sono liberi di apportare modifiche al codice per aggiungere funzionalità, rimuovere bug, e fare refactoring. A prima vista una tale libertà può sembrare pericolosa, tuttavia bisogna tener conto del fatto che per ogni unità esistono degli Unit Tests che ogni parte di codice modificato deve superare al 100% prima di poter essere integrata nel sistema.

Nella pratica, il collective code ownership è molto più sicuro che avere una singola persona che si occupa di gestire delle specifiche classi, in particolare se pensiamo al fatto che questa persona potrebbe lasciare il team.

LEAVE OPTIMIZATION TILL LAST.

Non ottimizzare mai fino alla fine! E' inutile cercare di capire in anticipo quale sara il collo di bottiglia del sistema, è molto più comodo misurarlo alla fine.

Make it work, make it right, then make it fast!

NO OVERTIME.

Evitare gli straordinari e le aggiunte di risorse umane per rientrare nei tempi stabiliti. Il codice scritto negli straordinari, specialmente a notte fonda, è di scarsa qualità. L'inserimento di nuovi programmatori crea spesso problemi e oneri non indifferenti di coordinamento e addestramento, in particolar modo se si è vicini alla data di rilasico.

TESTING

- All code must have unit tests.
- All code must pass all unit tests before it can be released.
- When a bug is found tests are created.
- Acceptance tests are run often and the score is published.

ALL CODE MUST HAVE UNIT TESTS.

Gli unit test in XP sono leggermente diversi da quelli tradizionali. Innanzitutto è necessario creare o scaricare uno unit test framework in modo da poter generare delle suite di test automatizzate, poi bisogna testare tutti i componenti del sistema ed infine, come già spiegato, bisogna creare i test prima del codice.
Al momento di integrare del codice, bisogna allegare anche il relativo codice di test. Se si scopre che manca uno unit test è necessario crearlo immediatamente. Avere dei test automatici disponibili per ogni componente o classe del sistema farà sicuramente risparmiare moltissimo tempo.
Gli unit test permettono il collective code ownership. Creando delle procedure di test ci si assicura che chi apporterà delle modifiche al codice avrà modo di verificare che esso funzioni. Gli unit test permettono inoltre il refactoring: si possono utilizzare per verificare che un cambiamento nella struttura non ha modificato le funzionalità di un componente.

ALL CODE MUST PASS ALL UNIT TESTS BEFORE IT CAN BE RELEASED.

Il codice deve ovviamente passare tutti gli unit test prima di poter essere rilasciato.

WHEN A BUG IS FOUND TESTS ARE CREATED.

Quando si incontra un bug devono essere creati dei test particolari in modo da evitare il riproporsi del baco. In particolare è necessario creare un acceptance test per il problema: questo permette al cliente di definire concisamente il problema e di comunicarlo ai programmatori. Questi infine hanno un test fallito su cui concentrarsi ed hanno un modo per poi verificare se il problema sia stato risolto o meno.
A partire dal test di accettazione i programmatori possono creare degli unit test per determinare il problema a più basso livello. Quando gli unit test verranno superati al 100% il software verrà sottoposto nuovamente all'acceptance test per verificare l'effettiva eliminazione del bug.

ACCEPTANCE TESTS ARE RUN OFTEN AND THE SCORE IS PUBLISHED.

I test di accettazione vengono creati a partire dalle user stories durante ogni iterazione. Il cliente specifica degli scenari di utilizzo tramite i quali verificare la corretta implementazione si una storia. Una singola storia può avere uno o più acceptance test di modo da poter verificarne la funzionalità. Ogni test di accettazione rappresenta una funzionalità del sistema. Il cliente è responsabile di controllare la correttezza di questi test e deve decidere, tra i test che falliscono, quali hanno la maggiore priorità. Una storia non è considerata completata finché non supera tutti i sui acceptance test. Questi test dovrebbero essere il più possibile automatizzati in modo da poter essere eseguiti spesso.

PROPRIETA' DI UN SOFTWARE

PROCESSO E PRODOTTO

Lo scopo è sviluppare un software, il processo è come lo facciamo. Sono entrambi importanti.

IL PRODOTTO SOFTWARE

E' diverso da altri tipi di prodotto.

- E' intangibile, difficile da descrivere e valutare.
- Malleabile, può essere modificato, diventare qualcosa di diverso.
- Prodotto ad alta intensità dal lavoro umano, non coinvolge nessun processo manifatturiero. (è prettamente creativo, la parte di duplicazione è la più semplice e meno importante)

INDICATORI DI QUALITA'

- **PROCESSO – PRODOTTO**
- **INTERNA – ESTERNA** (le qualità interne influiscono sulle esterne)

  ESEMPIO
  Interna: essere modulare Esterna: qualità dell'interfaccia

La qualità interna influisce su quella esterna, la qualità del processo influisce su quella del prodotto.

CORRETTEZZA

Il software è corretto se soddisfa le specifiche (voglio che il prodotto faccia ciò che mi aspetto).
Se le specifiche sono state date formalmente, la correttezza può essere definita formalmente. Può essere provata come un teorema o invalidata da contro-esempi (testing). Con il testing ho garanzie minori che con i metodi formali.
E' meglio provare a sviluppare un software corretto "a priori", e questo tramite opportuni processi (che vedremo in seguito) ed opportuni strumenti (linguaggi di alto livello, compilatori, strumenti di analisi, strumenti per la generazione automatica del codice, riutilizzo dell'esistente…)

I LIMITI DELLA CORRETTEZZA

E' una proprietà assoluta (si/no), non c'è il concetto di "grado di correttezza"
Inoltre la correttezza dipende dalle specifiche, cosa succede se le specifiche sono errate rispetto ai requisiti?

AFFIDABILITA' E ROBUSTEZZA

AFFIDABILITA'

Può essere definita come "probabilità di assenza di insuccessi per un certo periodo di tempo".
L'utente può "fidarsi" del software.
Se le specifiche sono corrette, tutto il software corretto è affidabile, ma non vale il viceversa.

ROBUSTEZZA

Il software appare "ragionevole" anche in situazioni non previste dalle specifiche (input errati, errori hardware…)

PRESTAZIONI

Uso efficiente delle risorse (memoria, tempo di elaborazione)

- Può essere verificata con metodi formali (analisi della complessità), o di tipo quantitativo (valutazione delle performance tramite modelli e simulazioni)
- Può incidere sulla scalabilità (in genere se una aumenta, l'altra diminuisce; ad esempio performance che possono andare bene su una piccola LAN possono non andare bene su grandi intranet)
- Può incidere sull'usabilità
- Può cambiare con la tecnologia

USABILITA'

E' una qualità esterna (cosa viene percepito dall'utente), non ci sono definizioni formali. Occorre definire chi è l'utente poiché l'usabilità varia in base a chi deve usare il sistema.

ALTRE PROPRIETA'

- Manutenibilità
- Riusabilità (simile alla manutenibilità, ma si intende quanto dell'applicazione posso riusare in un'altra applicazione)
- Portabilità (adattamento a diverso ambienti)
- Interoperabilità (cooperare con applicazioni diverse)

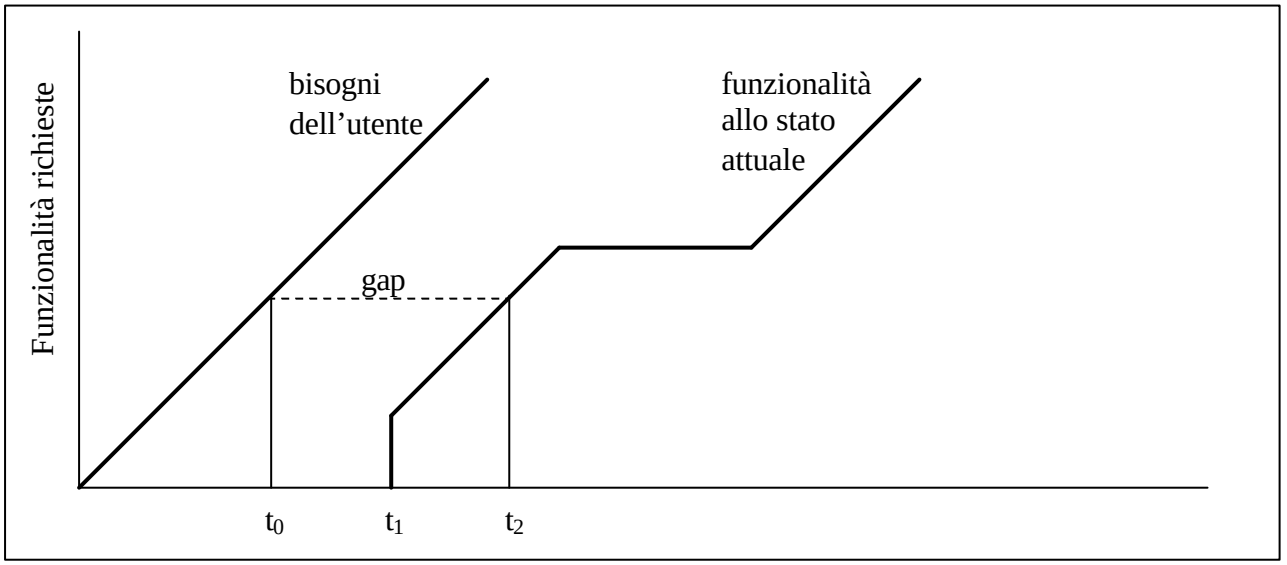
PROPRIETA' DEL PROCESSO

Produttività: si come "*item sviluppati per unità di sforzo.*"
numero di linee di codice prodotte da una persona in un mese. Ci sono dei limiti, infatti oltre alle linee di codice si fanno documentazione, test etc… e di questo non si tiene conto.

- **Unità di sforzo**: mesi uomo (da notare che 1 persona x 7 mesi ? 7 persone x 1 mese)
- **Item sviluppati**: linee di codice, funzioni

E' una misura dello sforzo ma non della qualità. Varia in base alla persona ma anche al gruppo di lavoro in cui una persona è inserita.

Tempestività: la capacità di reagire a richieste di modifica in maniera tempestiva



In t_0 abbiamo la necessità dell'utente (lineare), da quando si presenta la necessità a quando il progetto è utilizzabile passa del tempo (gap). Mentre sviluppo il progetto, posso anche avere fasi in cui non aggiungo funzionalità. Il processo deve essere strutturato in modo da gestire gli stop senza deteriorare le funzionalità.

PRINCIPI

- Rigore e formalità
- Anticipare il cambiamento
- Modularizzazione
- Astrazione (possibilità di creare modelli)
- Generalità
- Incrementalità