

# **APPUNTI DI INGEGNERIA DEL SOFTWARE**

**tratti dagli appunti del prof. Montangero, prof. Ambriola, prof. Cignoni e  
prof.ssa Semini  
Università di Pisa - A.A. 2003/2004**

**Realizzato da: Fabrizio Ciacchi - [fabrizio@ciacchi.it](mailto:fabrizio@ciacchi.it) - [fabrizio.ciacchi.it](http://fabrizio.ciacchi.it)  
Documento realizzato con [OpenOffice](#) e [AbiWord](#)  
Documento rilasciato sotto licenza **FDL****

***"Non si può cambiare il tempo con i programmatori"***

Frederick Brooks, manager del progetto OS/360

---

## LEZIONE 0 - PRESENTAZIONE DEL CORSO

Il compito dell'ingegneria del software è sostanzialmente quello di studiare le varie fasi di progettazione, sviluppo e mantenimento del software, fornendo delle analisi che ne possano assicurare la qualità.

Generalmente gli obiettivi di un corso di Ingegneria del Software sono:

- Inquadrare il contesto di riferimento;
- Identificare i problemi della produzione di software;
- Comprendere gli strumenti concettuali;
- Presentare soluzioni e metodi;
- Attenersi a standard di riferimento.

I temi trattati all'interno del corso stesso saranno, quindi:

- Il processo software;
- Controllo della qualità;
- Processi di supporto;
- Analisi e progettazione;
- Verifiche e prove

I testi consigliati per il corso sono:

- Lucidi (scaricabili sul sito di informatica dell'università di Pisa);
  - Syllabus (un'estratto di testi americani, tradotti e riassunti in questi appunti, disponibili a richiesta dal professore);
  - Testi di consultazione:
    - V. Ambriola, G.A. Cignoni, "Laboratorio di programmazione", Jackson Libri, 1996;
    - C. Ghezzi, "Ingegneria del software", Mondadori informatica, 1991;
    - G.A. Cignoni, P. De Risi, "Il test e la qualità del software", Il sole 24 ore, 1998.
-

# LEZIONE 1 - L'INGEGNERIA DEL SOFTWARE

## 1.0 - INTRODUZIONE

Che cos'è l'ingegneria del software? a questa domanda è difficile rispondere fornendo una sola definizione. Innanzitutto guardiamo gli argomenti trattati dalla materia, facendo una rapida carrellata dei temi principali di ingegneria del software.

L'ingegneria del software nasce e viene incontro alla produzione industriale organizzata di software, permettendo di produrre in grande, di assicurare la qualità dei prodotti e di garantire l'efficienza della produzione. I software commerciali, come sappiamo, possono essere suddivisi in quattro categorie:

- Sistemi software su commessa;
- Pacchetti software;
- Componenti software;
- Servizi su sistemi e dati.

Facendo una rapida carrellata possiamo vedere come la produzione di software sia notevolmente cresciuta negli ultimi 50 anni, basti pensare ai progetti spaziali della NASA, la sonda Mercury (1962) aveva un software di un milione di LoC (Line of Code, Linee di Codice, unità di misura poco indicativa della qualità ma sicuramente importante per capire l'entità di un software), il software dell'Apollo (1969) aveva circa 11 milioni di LoC, quello dello Shuttle (1981) era di 37 milioni, mentre quello del telescopio Hubble superava gli 80 milioni; oppure possiamo prendere come esempio la spesa dello stato italiano per la P.A., circa 2,56 miliardi di euro nel 2001.

Ma l'industria del software non è solo rose e fiori, anzi ci sono dei problemi da non sottovalutare, come i ritardi dei progetti (dovuti a difficoltà nelle fasi iniziali, cambi di piattaforma e difetti nel prodotto finale) e i fallimenti (a volte clamorosi come spiegato più avanti) dovuti ad obsolescenza prematura, incapacità di raggiungere gli obiettivi o l'esaurimento di fondi.

Per porre rimedio a tutto questo l'ingegneria del software cerca di studiare metodi per controllare il software ed il suo processo di produzione.

## 1.1 - IL SOFTWARE: PROCESSO E PRODOTTO ("Ingegneria del Software", Ghezzi, Capitolo 1)

Parlando dell'informatica e del software in generale, bisogna considerare che si tratta comunque di una famiglia di discipline (tra cui l'ingegneria del software) relativamente giovani; l'utilizzo del computer aumentò durante la seconda guerra mondiale per la sua straordinaria potenza di calcolo e di automazione di attività difficilmente eseguibili manualmente.

Infatti i primi utilizzatori sono essenzialmente fisici, matematici, ingegneri, ecc. che utilizzano il computer per eseguire complessi calcoli matematici, fisici, ecc.

Le caratteristiche più importanti di questa fase pionieristica sono:

- si utilizzano linguaggi di programmazione a basso livello (strumenti semplici);
- chi sviluppa software è un tecnico che si esprime in modo rigoroso e formalizzato;
- l'applicazione sviluppata non ha necessità di sopravvivere a lungo.

Questa situazione è andata modificandosi e grazie all'avvento dei "programmi gestionali" il computer ha potuto permettere la gestione, non più di soli numeri, ma anche di informazioni che vengono utilizzate in modo strategico per basare delle scelte economiche, gestionali, ecc. Grazie a questo sviluppo aumenta il divario tra chi produce software (programmatore) e chi lo utilizza (utente), in quanto l'utente non ha né il tempo né la voglia di imparare ad utilizzare gli strumenti di programmazione.

Analogamente la complessità e la criticità delle applicazioni cresce notevolmente ed è per questo che accanto agli EDP (Electronic Data Processing) interni delle aziende, nascono le software house. Si passa così dalla fase iniziale, individuale e creativa (software come arte) ad una fase caratterizzata da piccoli gruppi di esperti (software come artigianato) che producono software professionale.

Ed è proprio dopo questo passaggio che aumenta la quantità di software prodotto ed innesca riflessioni su come aumentare la produttività dello sviluppo e la qualità finale del prodotto. Si passa così gradualmente alla produzione di software come industria, intendendo un processo di sviluppo che coinvolge molte persone su progetti di grosse dimensioni e che deve essere pianificato ed automatizzato il più possibile.

Si pone quindi il problema di poter certificare la qualità dei prodotti, soprattutto se essi sono critici e complessi, attraverso uno sviluppo con metodologie efficaci e che aderisca a standard di produzione che rendano possibile lo scambio di personale.

Questa evoluzione (arte -> artigianato -> industria) venne concepita già nella seconda metà degli anni '60, ed infatti ad una conferenza NATO organizzata a Garmisch (Germania) nell'ottobre del 1968 venne coniato il

termine "ingegneria del software" proprio per testimoniare l'esigenza che la produzione di software diventasse una disciplina ingegneristica e quindi basata su solide basi teoriche e metodologiche che permettessero la progettazione, produzione e manutenzione di applicazioni che fornissero le caratteristiche di qualità previste mediante l'uso delle risorse previste. Ma si trattava più di una via di evoluzione piuttosto che dell'individuazione di una prassi da seguire, questo perché, contrariamente alle altre discipline ingegneristiche, che hanno decenni o secoli di basi teoriche e di metodologie empiriche consolidate, lo stato stesso della disciplina era molto rudimentale, ed anche oggi, sebbene si inizino a delineare i limiti di questa disciplina, siamo ancora in una fase iniziale di tale processo.

### **1.1.1 - LA DIMENSIONE ECONOMICA**

Non dobbiamo sottovalutare la dimensione economica della produzione di software, dagni anni '50 ad oggi il numero di persone coinvolte (e la corrispondente quantità di denare investito in tale area) non ha cessato di crescere nei paesi industrializzati. La questione però, posta al microscopio, rivela che ci sono dei problemi che non sono da sottovalutare. Innanzitutto i ritardi dei progetti che generano la domanda nascosta o inevasa (in inglese backlog), ritardi variabili tra i due ed i quattro anni. Il perché di questi ritardi fa capo a due problemi (legati tra loro): La carenza di personale specializzato, dovuta anche alle strutture formative che non sono in grado di preparare adeguatamente il numero necessario di specialisti del settore; la necessità di adibire una notevole quantità di addetti alla manutenzione del software esistente. Questi due problemi sono strettamente collegati, in quanto per molte aziende il successo (o la continuazione stessa) dell'attività è strettamente legato alla propria struttura informatica, e spesso si fa ricorso a persone poco preparate o addirittura inadeguate (nda: incompetenti). E come il proverbiale cane che si morde la coda, questo genera l'immissione di cattivo software sul mercato (cattivi programmatori sono la causa di cattivo software) che necessita di notevoli sforzi di manutenzione. Purtroppo l'industria del software non ha saputo reagire adeguatamente. Mentre in altri settori la produzione è diventata per lo più automatizzata (si pensi, ad esempio, all'hardware), nel software questo non accade, anche perché la fase di progettazione di un software è brain intensive (ad alta intensità di cervello) e, quindi, difficilmente meccanizzabile.

Può risultare interessante domandarsi quale possa essere la produttività media delle persone nello sviluppo del software. La difficoltà di rispondere a questa domanda non fa altro che confermare che ci sono aspetti dell'ingegneria del software che ancora non conosciamo in maniera scientifica, e per i quali non esiste una teoria interpretativa universalmente accettata.

Un modo semplicistico di valutazione può essere calcolare le linee di codice prodotte (line of code, LoC) per giorno da una data persona. L'esperienza mostra che ci si assesta ad un valore di qualche decina (10-30), valore che può diventare un decimo nel passaggio da software di tipo sequenziale tradizionale a software critico, concorrente e in tempo reale.

### **1.1.2 LA DIMENSIONE SOCIALE**

Analizzando lo sviluppo dell'hardware e del software degli ultimi 30 anni sembra che quest'ultimo sia andato lentamente. In realtà è lo sviluppo dell'hardware che è andato incredibilmente veloce, basti pensare che ha guadagnato 6 ordini di grandezza nel rapporto prestazioni/costo. Il software è cresciuto notevolmente ma si scontra con la sua complessità intrinseca e cioè con il fatto che "a parità di dimensione, la complessità di capire, descrivere, progettare un frammento di software è di gran lunga superiore a quella corrispondente per qualunque altro prodotto. Programmi di grande complessità non sono ottenuti per giustapposizione di parti identiche o simili, o ben isolate tra di loro, come in gran parte dei progetti industriali; al contrario, raramente si ravvisa una qualche regolarità nella struttura di un programma".

A queste difficoltà intrinseche si aggiungono difficoltà dovute ad approcci inadeguati, scarsamente basati su fondamenti metodologicamente rigorosi, che rendono l'affidabilità del software bassa rispetto alle esigenze delle applicazioni. Queste ultime, per contro, tendono a diventare sempre più complesse e critiche. Basti pensare ad applicazioni del software che ledono la vita reale, come software per la gestione economica, per il controllo degli aerei, per i sistemi di monitoraggio di pazienti o sistemi per le applicazioni militari.

Queste considerazioni rendono evidente, da un lato, la necessità di formare ingegneri del software che siano in grado di progettare applicazioni sofisticate sfruttando al meglio il bagaglio di conoscenze teoriche e metodologiche che si sono sviluppate nel corso degli ultimi anni. Ma va fatto notare che l'ingegnere del software pone questa professionalità al servizio della propria coscienza civile, secondo la quale il problema della qualità delle applicazioni va sempre messo in primo piano. Bisogna sempre procedere con metodologia e professionalità senza farsi prendere dai facili (e fasulli) entusiasmi secondo i quali il software può facilmente risolvere in modo magico qualsiasi problema. Bisogna, inoltre, avere molta umiltà nell'affrontare problemi la cui complessità va al di là dei limiti imposti dalla tecnologia e se possono generare rischi di qualche genere, questi devono essere presi in serio esame e devono essere effettivamente controllati a posteriori.

E' molto importante pensare all'episodio di D.L. Parnas, uno dei massimi esperti di ingegneria del software, che ha motivato pubblicamente le proprie dimissioni dalla commissione del progetto "star wars" (lo scudo spaziale

voluto da Reagan) dichiarando che le tecnologie non sono in grado (e non lo saranno in un futuro prossimo) di affrontare le problematiche di enorme complessità e criticità di applicazioni simile a quelle del programma guerre stellari.

### **1.1.3 - 1968: CONFERENZA DI GARMISH**

Ecco un estratto della conferenza di Garmish del 1968, conferenza nella quale venne coniato il termine "Ingegneria del software".

#### **1.1.3-a - BACKGROUND OF CONFERENCE**

Nel 1967 il comitato scientifico della NATO iniziò a concentrarsi molto sullo sviluppo del software e pensò di formare un istituto internazionale informatico. Nell'autunno del 1967 venne stabilito un gruppo di studio sulle discipline informatiche che doveva ricoprire tutti i campi dell'informatica ed elaborare suggerimenti per il comitato scientifico.

Il gruppo di studio si concentrò su quelle aree che necessitavano di supporto nazionale od internazionale, in particolare pose la sua attenzione sullo sviluppo del software, infatti propose l'organizzazione di una conferenza che avesse come tema "l'ingegneria del software". La frase "ingegneria del software" fu scelta deliberatamente in modo provocatorio, per sottintendere il bisogno, nello sviluppo del software, di metodologie pratiche e teoriche comuni in altri settori dell'ingegneria. Fu suggerito che 50 tra i massimi esperti di discipline concernenti lo sviluppo del software fossero invitati per presiedere in modo attivo alla conferenza, concentrandosi su tre aspetti fondamentali: Design (???) di Software, Produzione di Software e Servizi del Software.

Vennero scelti dei leader per ogni settore e fu scelto Il Dr. Arnth-Jensen, della Divisione affari scientifici della NATO, per presiedere all'organizzazione della conferenza.

Nel Marzo del 1968 si incontrarono a Brussels i leader di ogni gruppo ed il gruppo di studio per i dettagli finali della conferenza.

La conferenza fu pensata per studiare i problemi dell'ingegneria del software e per discutere sulle possibili tecniche, metodi e sviluppo che avrebbero potuto porre una soluzione. Si sperò di poter identificare le necessità presenti, quelle a breve termine e quelle a lungo termine per aiutare sia i produttori che gli utenti.

#### **1.1.3-b - SOFTWARE ENGINEERING AND SOCIETY**

Adesso presentiamo un breve estratto con le frasi più importanti della conferenza.

##### **LA PERCENTUALE DI CRESCITA DEL SOFTWARE**

D'Agapeyeff: Nel 1958 un comune produttore di computer europeo aveva meno di 50 programmatori, oggi ne ha circa 1000-2000; di quanti ne avrà bisogno nel 1978?

##### **LA CRESCITA VISTA CON PREOCCUPAZIONE**

David: Le fasi di ricerca, sviluppo e produzione di software, spesso sono unite in un processo unico. Separare queste componenti ed avere un approccio di lavoro a step ci permette di lavorare con meno rischi e maggiori successi. Ma ci sono buoni motivi per credere che lo sviluppo di software che include concetti nuovi implichi rischi non solo incalcolati, ma anche incalcolabili.

##### **MA L'INDUSTRIA DEL SOFTWARE HA ANCHE I SUOI SUCCESSI**

Buxton: Il 99% dei computer lavora con una tollerabilità soddisfacente. Ci sono migliaia di rispettabili installazioni Fortran-oriented che usano differenti macchina, e molte buone applicazioni di data-processing lavorano senza problemi.

##### **COMUNQUE CI SONO AREE VISTE CON PARTICOLARE SCETTICISMO**

David and Fraser: Particolarmente allarmante sembra essere l'indesiderata fallibilità del software di grandi dimensioni, considerando che un malfunzionamento in sistemi hardware-software avanzati può fare la differenza tra la vita e la morte.

##### **SONO TUTTI D'ACCORDO CHE L'INGEGNERIA DEL SOFTWARE E' AGLI STATI INIZIALI**

Graham: Oggi si tende ad andare avanti per anni, con tremendi investimenti per vedere sistemi, di cui non si capisce perchè siano stati cominciati, che non lavorano come promesso. Noi costruiamo sistemi, così come i fratelli Wright costruivano aeroplani; costruiamo le cose principali, lasciamo che vadano, le facciamo rompere (crashare) e ricominciamo da capo.

##### **OVVIAMENTE OGNI CAMPO HA I SUOI PROBLEMI DI CRESCITA**

Oillette: Siamo nell'analogia situazione dell'industria degli aircraft, che ha i nostri stessi problemi nel produrre sistemi in modo sistematico e seguendo determinate specifiche. Noi forse abbiamo

maggiori esempi di grandi sistemi fatti male rispetto a quelli fatti bene, ma siamo un'industria giovane e stiamo imparando come fare meglio.

## **IL PROBLEMA PIU' GRANDE E' LA PRESSIONE A PRODURRE SOFTWARE SEMPRE PIU' GRANDE E COMPLESSO**

Buxton: Ci sono pressioni estremamente forti sui produttori, sia da parte degli utenti sia da parte di altri produttori. Alcune di queste pressioni, che contribuiscono ai nostri problemi, sono chiare. Ad esempio, l'incremento del traffico aereo in Europa influisce sul fatto che ci sono pressioni per la creazione di un sistema di controllo automatizzato.

## **E PER FINIRE ...**

Gill: E' molto importante che tutti i responsabili per grandi progetti che coinvolgono l'uso del computer devono prestare attenzione al fatto che l'uso del software per scopi poco lontani da quelli che lo stato attuale della tecnologia ci consente, permette di tollerare i rischi coinvolti.

### **1.1.4 - DICHIARAZIONE DI D.L. PARNAS**

Tratto dal New York Times, 7/12/85, pagina A6 (traduzione); in merito all'SDI (STRATEGIC DEFENSE INITIATIVE) conosciuto anche come "Star Wars" (o Guerre Stellari in italiano):

*"Washington, 11 luglio - Uno scienziato informatico si è dimesso da una commissione di difesa antimissilistica, sostenendo che non sarà possibile programmare in modo sicuro un sistema di computer per la gestione di una battaglia o di assumere che funzioneranno quando saranno soggetti all'attacco di missili nucleari.*

*Lo scienziato, David L. Parnas, un professore dell'università di Victoria (Columbia inglese) che è consigliere dell'Ufficio di Ricerca Navale di Wasghinton, è stato uno dei nove scienziati richiesti dall'uffici di Iniziative di Difesa Strategiche per creare, al costo di 1000\$ al giorno, un sistema informatico in supporto alla gestione della battaglia.*

*Il professor Parnas ha detto in una lettera di licenziamento e 17 memorandum allegati che non sarà mai possibile testare in modo realistico la grande quantità di computer che dovrebbero collegare e controllare un sistema di sensori, armi antimissilistiche, dispositivi di guida e supporto, e stazioni di gestione della battaglia.*

*Inoltre, dice Parnas, non sarebbe possibile seguire le pratiche ortodosse di scrittura dei programmi nei quali errori e bug vengono scovati ed eliminati nell'uso giornaliero prolungato.*

*- Io Credo - sostiene Parnas - che sia nostro dovere, come scienziati ed ingegneri, rispondere che non abbiamo magie tecnologiche che permettano di fare ciò. Il presidente (Reagan) ed il pubblico lo deve sapere - .*

*Nei suoi memorandum, il professore espone in maniera dettagliata i suoi dubbi. Egli intuisce che programmi su larga scala diventano affidabili solo attraverso modifiche basate su un uso realistico. Egli marca come non realistica l'idea che computer, intelligenza artificiale o simulazioni matematiche possano risolvere il problema. Altri scienziati hanno espresso i propri dubbi sul fatto che possano essere scritti programmi su larga scala esenti da errori. Herbert Lin, un ricercatore al MIT, ha detto che la lezione più semplice è che - nessun programma lavora correttamente la prima volta -."*

## **1.2 - DEFINIZIONI DI "INGEGNERIA DEL SOFTWARE"**

- Definizione dell'IEEE:

***L'approccio sistematico allo sviluppo, all'operatività, alla manutenzione ed al ritiro del software.***

E' un approccio di tipo sistematico da cui si evince che il software è un prodotto con il suo ciclo vitale.

- Definizione di Fairley:

***La disciplina tecnologica e gestionale per la produzione sistematica e la manutenzione di prodotti software sviluppati e modificati con tempi e costi preventivati.***

Con questa definizione si introduce il concetto di tipo gestionale (costi, tempi, risorse) e del controllo della qualità (costi e risultati definiti).

Volendo cercare una definizione unica, potremmo dire che l'ingegneria del software è un corpus di teorie, metodi e strumenti (di tipo sia tecnologico sia organizzativi) che consentano di produrre applicazioni con le desiderate

caratteristiche di qualità. Il suo obiettivo è quello di mettere in grado lo specialista di software di affrontare la complessità dei grossi progetti gestendo in modo produttivo le risorse, specie quelle umane, e sviluppando prodotti che rispecchiano le caratteristiche desiderate di qualità entro i vincoli economici e di tempo specificati. Ne deriva che l'ingegneria del software comprende un insieme di contenuti di tipo teorico di base, un insieme di metodi rigorosi che trovano giustificazione nelle basi teoriche, un insieme di tecnologie concrete da impiegare nei processi produttivi, il tutto cementato da una sensibilità pratica di tipo empirico che sia in grado di calare teorie, metodi e strumenti in un atteggiamento progettuale capace di calibrare l'approccio in funzione del problema specifico da risolvere.

### **1.3 - TEMI DI INGEGNERIA DEL SOFTWARE**

- REALIZZAZIONE DI SISTEMI SOFTWARE
  - STRATEGIE DI ANALISI E PROGETTAZIONE
    - Tecniche per la comprensione e la soluzione di un problema;
    - Top-down, bottom-up, progettazione modulare, OO.
  - LINGUAGGI DI SPECIFICA E PROGETTAZIONE
    - Strumenti formali per la definizione di sistemi software;
    - Reti di Petri, Z, OMT, UML.
  - AMBIENTI DI SVILUPPO
    - Strumenti per analisi, progettazione e realizzazione;
    - Strumenti tradizionali, CASE, CAST e RAD.
- PROCESSO SOFTWARE
  - ORGANIZZAZIONE E GESTIONE DEI PROGETTI
    - Metodi di composizione dei gruppi di lavoro;
    - Strumenti di pianificazione, analisi, controllo.
  - CICLI DI VITA DEL SOFTWARE
    - Definizione e correlazione delle attività;
    - Modelli ideali di processo di sviluppo.
  - MODELLI DEL PROCESSO DI SVILUPPO
    - Norme per la definizione delle attività;
    - Strumenti per la definizione dei processi.
- QUALITA' DEL SOFTWARE
  - METODI DI VERIFICA E CONTROLLO
    - Metodi di verifica, criteri di progettazione delle prove;
    - Controllo della qualità, valutazione del processo di sviluppo.
  - MODELLI DI QUALITA'
    - Definizione di caratteristiche della qualità;
    - Valutazione dei prodotti.
  - METRICHE DI SOFTWARE
    - Unità di misura, scale di riferimento, strumenti;
    - Indicatori di qualità.

### **1.4 - CASI DI STUDIO**

Presentiamo qui di seguito alcuni casi di studio su clamorosi fallimenti di noti progetti software:

#### **LONDON AMBULANCE SERVICE**

Era stato creato un sistema automatizzato per gestire il servizio delle ambulanze che derivava dall'unificazione di 3 servizi diversi; doveva esserci l'ottimizzazione dei percorsi, la guida vocale degli autisti, la gestione delle chiamate, i backup, la localizzazione in tempo reale delle ambulanze, lo stato dei pazienti e molto altro.

Il software purtroppo era progettato male, con errori, senza sistemi di controllo in caso di errori, con l'impossibilità di fare i backup, la mancata preparazione del personale, la mancata conoscenza (da parte dei progettisti) del problema; ed è costato, non solo vite umane, ma anche 11 milioni di euro; l'ultima versione è stata abbandonata dopo soli 3 giorni. Si può dire che questo è uno dei disastri più "eclatanti"

nella storia del software e che va preso come esempio su "come non progettare un software".

### **AEROPORTO DI DENVER**

Fonte CNET.com - 1995:

*"Il nuovissimo Denver International Airport doveva essere all'avanguardia, con il suo complesso sistema computerizzato di smistamento dei bagagli e ottomila chilometri di cavi in fibra ottica. Ahimé, dei difetti nel sistema di smistamento finirono per tritare alcune valigie e far schiantare contro i muri i carrelli automatici. L'aeroporto fu costretto ad aprire 16 mesi dopo la data prevista, con uno sfioramento di 3,2 miliardi di dollari rispetto ai preventivi... e con un sistema di smistamento bagagli quasi interamente manuale."*

Il sistema doveva essere completamente automatizzato, con 35 Km di rete, 4000 carrelli, 5000 "occhi" e 56 lettori. Investimento di 193 milioni di \$; le perdite sono state stimate in circa 1 milione di \$ al giorno (costi + mancati guadagni).

### **ARIANE 5**

Il 4 giugno 1996 viene lanciato per la prima volta il vettore Ariane 5, punta di diamante del programma spaziale europeo. Dopo 39 secondi di volo interviene il sistema di autodistruzione, trasformando l'Ariane 5 e il suo carico pagante (quattro satelliti scientifici non assicurati) in quello che è stato definito "il più costoso fuoco d'artificio della storia". Non ci sono vittime, dato che il missile non ha equipaggio e i frammenti dell'esplosione cadono in una zona disabitata della Guiana francese, ma i danni economici sono ingentissimi (1 miliardo di euro). Il disastro avviene perché un programma del sistema di navigazione tenta di mettere un numero a 64 bit in uno spazio a 16 bit. Il sistema, progettato per l'Ariane 4 e riciclato perché dimostratosi affidabile, tenta infatti di convertire la velocità laterale del missile dal formato a 64 bit al formato a 16 bit. Tuttavia l'Ariane 5 vola molto più velocemente dell'Ariane 4, per cui il valore della velocità laterale è più elevato di quanto possa essere gestito dalla routine di conversione, che a differenza di molte altre routine di conversione a bordo è priva dei normali controlli che evitano problemi di gestione.

Risultato: overflow, spegnimento del sistema di guida, e trasferimento del controllo al secondo sistema di guida, che però essendo progettato allo stesso modo è andato in tilt nella medesima maniera pochi millisecondi prima. Privo di guida, il vettore si autodistrugge.

Ironia della sorte, la funzione di conversione difettosa che causa lo spegnimento del sistema di guida non ha alcuna utilità pratica una volta che l'Ariane è partito: serve soltanto per allineare il vettore con le coordinate celesti (l'Ariane ha un sistema di navigazione inerziale). Ma i progettisti avevano deciso di lasciarla attiva per i primi 40 secondi di volo in modo da facilitare il riavvio del sistema se si verificava una breve interruzione nel conto alla rovescia.

Da un articolo del Sole 24 Ore:

*"Appena nove secondi dopo il decollo, le due centrali inerziali già non riuscivano ad avere un allineamento corretto. Al 37° secondo le due centrali si dichiaravano contemporaneamente in situazione d'errore irrecuperabile e lanciavano una procedura di reset (...) realizzabile fino a 40 secondi dal via. (...) veniva assunta come base di riferimento la posizione istantanea del vettore, non coincidente con la posizione verticale sulla base di lancio, e il calcolatore ordinava una brusca correzione di rotta, portando a fine corsa gli ugelli sia dei booster che del motore criogenico. L'improvviso aumento degli sforzi aerodinamici portava a instabilità strutturale ("buckling") e il vettore si rompeva poco al di sotto dell'ogiva. Subito dopo si sono attivati i sistemi automatici di autodistruzione".*

Fonti: Ariane 5: errore nel software, Claudio Borgonovi, Sole 24 Ore, luglio 1996; A Bug and a Crash, James Gleick, <http://www.around.com/ariane.html>

### **PATRIOT**

Un errore nel sistema di guida, molto probabilmente sul controllo delle traiettorie, ha fatto sì che un missile patriot abbia colpito una caserma americana. Si pensa che l'errore non sia tanto nel software, ma proprio progettuale, cioè che mancasse proprio il controllo della traiettoria.

### **THERAC 25**

Estratto (tradotto) dal The Boston Globe, 20 Giugno 1986:

*"Una serie di emissioni accidentali spropositate di radiazioni da macchine identiche per la terapia del cancro, in Texas ed in Georgia, hanno ucciso una persona ed hanno lasciato altre due persone con grandi bruciature e paralisi parziale, secondo quanto riferito dagli investigatori federali.*

*Evidentemente causate da un errore nel programma che controlla i dispositivi, le emissioni (ignorate fino ad adesso) si crede che siano il peggior incidente medico conosciuto fino ad oggi.*

*I malfunzionamenti si sono presentati uno lo scorso anno e due a Marzo ed Aprile di quest'anno in due*

degli acceleratori lineari di fattura Canadese, venduti con il nome di Therac 25."

### **MARS CLIMATE ORBITER & MARS POLAR LANDER**

Tratto da La Repubblica del 28 Marzo 2000:

*"Il giudizio non è inatteso, visto che le ultime due spedizioni verso Marte si sono concluse con un disastro totale. Alla fine di settembre, il satellite Mars Climate Orbiter si è incendiato nell'atmosfera marziana per colpa di un assurda confusione tra metri e pollici. E nemmeno tre mesi dopo, in dicembre, la sonda Polar Lander è scomparsa per sempre pochi secondi dopo aver iniziato la discesa verso la superficie di Marte. Una perdita complessiva di quasi 300 milioni di dollari, ma soprattutto un crollo di credibilità, che ha immediatamente scatenato le accuse nei confronti dell'intera gestione del programma. Noto, e sbandierato, con lo slogan "cheaper, faster, better", economico, veloce e migliore, il programma potrebbe rivelarsi il più gigantesco boomerang della storia della Nasa dai tempi dell'esplosione del Challenger, nel 1986".*

### **MARS SPIRIT (27 gennaio 2004)**

Secondo quanto dichiarato al link <http://spaceflightnow.com/mars/mera/040126spirit.html>, il rover mars spirit sarebbe andato in crash per colpa di una variante dell'errore "fixed lenght buffer". Sembra che spirit abbia cercato di riavviarsi circa sessanta volte; come dice la legge di Woodhead: "Più lontano sei dal tuo server, più facilmente andrà in crash". La prossima volta mandiamo su anche un sistemista ;)

### **IL ROBOT KILLER: Riassunto**

La storia del Robot Killer è una storia inventata per far vedere come la non osservanza di regole etiche nello sviluppo del software possa uccidere una persona. Il racconto è formato da sette articoli di quotidiano, un articolo di giornale ed una intervista. Riassumendo la vicenda:

Il Robot Robbie CX30 prodotto dalla Silicon Techtronics uccide un operatore, a causa di oscillazioni improvvise. Da questo momento inizia una sorta di indagine, non solo da parte delle forze dell'ordine, ma anche dai giornali che raccontano la vicenda e dall'Università di Silicon Valley.

Si viene a sapere che (in ordine sparso):

- JANE ANDERSON viene licenziata dopo essersi opposta all'uso del modello a cascata per la costruzione del Robot.
- RANDY SAMUELS, il programmatore del codice che controllava il robot Robbie CX30, non aveva implementato nel robot le adeguate equazioni fisiche, aveva rubato del codice e non lo aveva adeguatamente testato.
- CINDY YARDLEY aveva falsificato i test del software per coprire il lavoro dei propri colleghi.
- MAX WORTHINGTON, un capo della Silicon Techtronics, aveva monitorato le comunicazioni dei propri dipendenti.
- SAM REYNOLDS, progettista in data processing, era stato messo a capo del progetto del Robbie CX30.
- MICHAEL WATERSON, presidente e CEO della Silicon Techtronics, aveva messo SAM REYNOLDS a capo del progetto.

Si può vedere da questi pochi punti che tutta una serie di pressioni economiche, politiche, personali e la non osservanza dei diritti di privacy, copyright ed usabilità abbiano portato alla morte di una persona. A nostro avviso ogni personaggio risulta coinvolto nell'uccisione dell'uomo e sono tutti egualmente responsabili.

---

## LEZIONE 2 - IL CICLO DI VITA DEL SOFTWARE

### 2.0 - INTRODUZIONE

Bisogna capire innanzitutto che un software ha un proprio ciclo di vita che consiste nella concezione (l'idea), lo sviluppo (la programmazione), la gestione (correzione errori, supporto ai clienti) ed il ritiro (smettere il supporto, fare una nuova versione) del software.

La modellazione di un ciclo di vita del software passa attraverso l'identificazione delle attività (lo sviluppo di modelli generici e quindi indipendenti dal prodotto; la decomposizione e l'uso della terminologia adatta per descrivere il problema) e l'organizzazione delle attività (che vanno ordinate e per le quali vanno stabiliti dei criteri per terminare un'attività e passare alla successiva).

E' importante porsi degli obiettivi e dei limiti ai quali il modello del ciclo di vita deve adeguarsi per fare un passo verso il processo definito. Innanzitutto bisogna porsi come obiettivo il controllo della qualità, che definisce il processo produttivo, identifica le attività, gli obiettivi e le dipendenze, e viene usato come strumento di pianificazione e gestione dei progetti. Il ciclo di vita di un software non è un metodo di sviluppo software (è infatti indipendente dalla piattaforma) e non è nemmeno un insieme di indicazioni e strumenti (non spiega come e con cosa operare).

Il primo modello conosciuto è il CODE&FIX (tradotto Programma e Correggi, va bene per aziende piccole) che possiamo definire un non-modello, in quanto le attività non sono né identificate né organizzate, ed i progetti sono non-gestiti (o gestiti male).

I modelli conosciuti sono quello a cascata, quello incrementale, quello evolutivo e quello a spirale. Il documento principale per capire i cicli di vita del software è l'ISO/IEC 12207.

### 2.1 - MODELLI DEL CICLO DI VITA DEL SOFTWARE

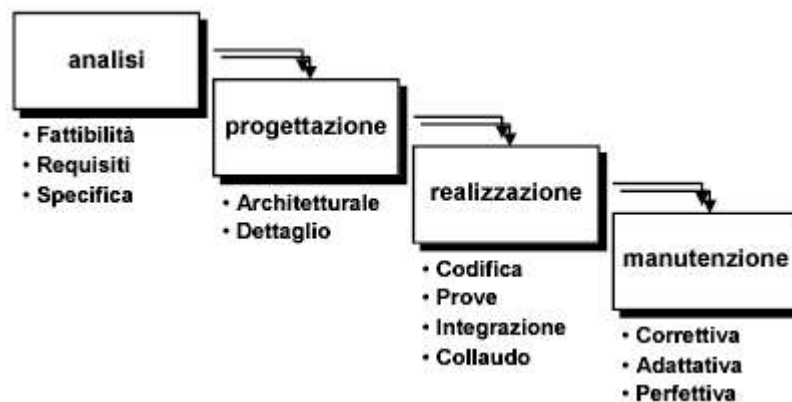
Ci sono diversi modelli del ciclo di vita del software. Comunque ce ne sono tre fondamentali e sono:

- A cascata (Waterfall);
- Modelli iterativi;
  - Incrementale (Incremental);
  - Evolutivo (Evolutionary);

Ognuno di questi cicli di vita può essere usato così com'è, oppure possono essere combinati per creare un modello ibrido. E' Attraverso il modello scelto che i processi, le attività ed i task dell'ISO/IEC 12207 sono collegati, e le loro relazioni di precedenza sono definite.

Cercheremo di spiegare i tre principali modelli del ciclo di vita del software con i loro rischi ed i loro benefici. I rischi ed i benefici devono essere presi in seria considerazione quando si sceglie un modello del ciclo di vita per un progetto.

#### 2.1.1 - A CASCATA (WATERFALL)



Il metodo a cascata è essenzialmente un metodo in cui si eseguono i passaggi uno alla volta, ed una sola volta, che sono:

- Determinare i bisogni dell'utente (di solito colui che paga);
- Definire i requisiti;

- Disegnare il sistema;
- Fabbricare il sistema;
- Fase di test;
- Fase di correzione;
- Consegna od uso.

In questo approccio ogni parte del software viene sviluppata, le attività ed i compiti nel processo di sviluppo sono usualmente eseguiti in serie. Comunque possono essere sviluppate parzialmente in parallelo quando attività sequenziali collidono.

Quando diverse parti del software sono sviluppate concorrentemente, le attività ed i compiti nel processo di sviluppo possono essere eseguite in parallelo rispetto a tutte le entità del software. Il processo di Manutenzione e quello Operativo sono sviluppati dopo il processo di sviluppo. I processi di Acquisizione, Approvvigionamento, Supporto ed Organizzativo sono eseguiti in parallelo con il processo di sviluppo.

### RISCHI

- I requisiti possono non essere compresi correttamente;
- Sistema troppo grande per svilupparlo tutto in una volta;
- Cambiamento rapido delle tecnologie;
- Cambiamento rapido dei requisiti;
- Risorse/Staff/Fondi limitati nell'immediato;
- Un prodotto intermedio non è utilizzabile.

### VANTAGGI

- Tutte le capacità del sistema vengono fuori tutte in una volta;
- E' necessario far fuori un vecchio sistema tutto in una volta.

Definito nel 1970 da Royce, questo modello è composto dalla successione di fasi sequenziali, in cui risulta impossibile tornare indietro; nel caso ci fossero errori nello studio dei requisiti si riparte da capo. Ogni fase viene caratterizzata dalla produzione di documenti che concretizzano la fase e che sono necessari per la fase successiva, non a caso questo modello viene considerato "document driven". Ogni fase risulta descritta in termini di attività e prodotti intermedi, contenuti e struttura dei documenti, responsabilità e ruoli coinvolti, e scadenza di consegna dei documenti; in questo modo si instaurano dipendenze causali e temporali e si ha la possibilità di identificare le attività per ogni fase. Purtroppo però questo modello manca di flessibilità, perchè non solo è impossibile tornare indietro (e quindi modificare qualche requisito), ma genera molta manutenzione, è burocratico e poco realistico; esistono quindi delle varianti di questo modello, che sono la cascata con prototipazione e con ritorni.

### APPROFONDIMENTO: I MODELLI ITERATIVI

I modelli iterativi sono caratterizzati dalla possibilità di adattarsi ai cambiamenti, come i cambiamenti di soluzioni e tecnologie, o i cambiamenti dei requisiti del committente. In linea generale si decompone la realizzazione del sistema e si ritarda la realizzazione delle componenti critiche, facendo si che le iterazioni siano pianificate.

#### 2.1.2 - INCREMENTALE (INCREMENTAL)



Il modello incrementale, anche chiamato "miglioramento del prodotto pianificato", comincia con una serie di

requisiti dati e si sviluppa in una sequenza di passi (o, tradotto dall'inglese, costruzioni). Il primo passo incorpora una parte dei requisiti, il passo successivo aggiunge un'altra parte dei requisiti, e così via, fino a quando il sistema è completo. Ad ogni passo i processi, le attività ed i compiti necessari vengono eseguiti; ad esempio l'analisi dei requisiti e l'analisi architettonica viene fatta una volta sola, mentre il disegno in dettaglio del software, la scrittura ed il test del codice, i test di integrazione e qualifica del software vengono eseguiti ad ogni passo.

Con questo approccio, come viene sviluppato un passo, le attività ed i compiti nel processo di sviluppo sono seriali o parzialmente paralleli. Quando passi consecutivi vengono sviluppati con una parziale concorrenza di attività e compiti nel processo di sviluppo, possono operare in parallelo.

Le attività ed i compiti nel processo di sviluppo sono ripetuti nella medesima sequenza in tutti i passi. I processi di Manutenimento ed Operativi possono operare in parallelo con il processo di Sviluppo. I processi di Acquisizione, Approvvigionamento, Supporto ed Organizzativo sono usualmente operativi in parallelo al processo di Sviluppo.

### **RISCHI**

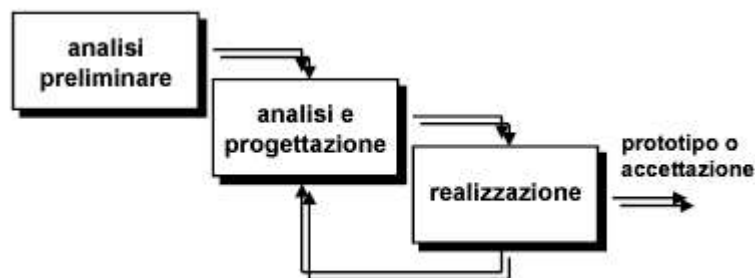
- I requisiti possono non essere compresi correttamente;
- Si preferiscono le capacità tutte in una volta; (???)
- Cambiamento rapido delle tecnologie;
- Cambiamento rapido dei requisiti;
- Risorse limitate a lungo termine.

### **VANTAGGI**

- Un prodotto intermedio è utilizzabile;
- Si necessita di capacità immediate. (???)
- Il sistema è partizionato naturalmente in incrementi;
- Le risorse possono essere incrementali.

E' da far notare che, con questo metodo, c'è l'ACCETTAZIONE DI VERSIONI PARZIALI, prima di passare al passo successivo dello sviluppo.

### **2.1.3 - EVOLUTIVO (EVOLUTIONARY)**



Anche Il modello evolutivo sviluppa il sistema in una serie di passi, ma differisce dal modello incrementale nel sapere che i requisiti non sono tutti compresi e non possono essere definiti inizialmente. In questo approccio, i requisiti sono parzialmente definiti in modo grezzo, quindi vengono raffinati in ognuno dei passi successivi. In questo approccio, quando ogni passo è sviluppato, le attività ed i compiti nel processo di sviluppo possono operare in serie o in parallelo. Le attività ed i compiti del processo di sviluppo usualmente operano ripetutamente nella stessa sequenza per tutti i passi. I processi di Manutenimento ed Operativi possono operare in parallelo con il processo di Sviluppo. I processi di Acquisizione, Approvvigionamento, Supporto ed Organizzativo usualmente operano in parallelo con il processo di Sviluppo.

### **RISCHI**

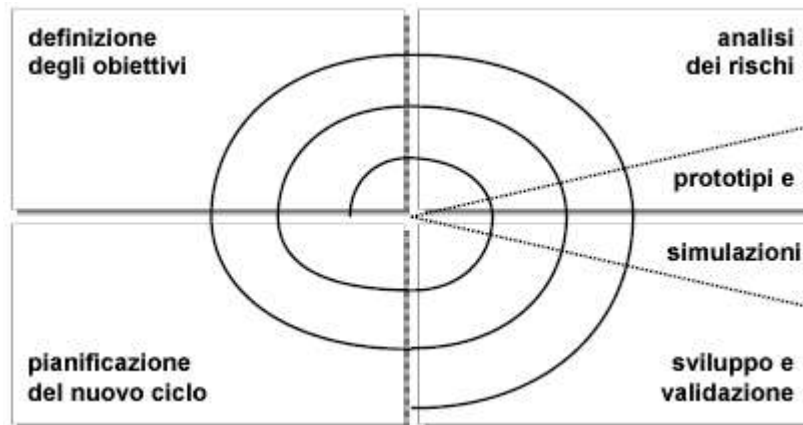
- Si preferiscono le capacità tutte in una volta; (???)

- Risorse limitate a lungo termine.

### VANTAGGI

- Si necessita di capacità immediate. (???)
- Un prodotto intermedio è utilizzabile;
- Il sistema è partizionato naturalmente in incrementi;
- Le risorse possono essere incrementali;
- Si necessita dei pareri dell'utente per capire tutti i requisiti;
- Il monitoraggio dei cambiamenti di tecnologia è facilitato.

## 2.2 - A SPIRALE (SPIRAL)



Il modello a spirale venne proposto nel 1988 da Boehm, infatti prevede quattro attività principali:

- Definizione degli obiettivi;  
Requisiti, identificazione dei rischi, piano di gestione;
- Analisi dei rischi;  
Studio di conseguenze e alternative, prototipi e simulazioni;
- Sviluppo e validazione  
Realizzazione del prodotto;
- Pianificazione;  
Decisione circa il proseguimento, pianificazione del ciclo.

E' più un protomodello, un modello astratto che deve essere specializzato. Infatti con questo modello vengono evidenziati gli aspetti gestionali, attraverso una pianificazione delle fasi e l'analisi dei rischi (modello **risk driven**).

Evidenzia i ruoli di committente e fornitore, il committente deve, ad esempio, dare definizione degli obiettivi, e indicazioni sulla pianificazione; il fornitore deve intervenire nello sviluppo e la validazione; entrambi devono valutare i rischi reali.

Si può dire che questo modello cerca di studiare, attraverso l'analisi dei rischi, quale sia, volta volta, la soluzione migliore, studiando soluzioni sempre nuove e che possano diminuire in maniera concreta e reale i rischi a cui si va incontro.

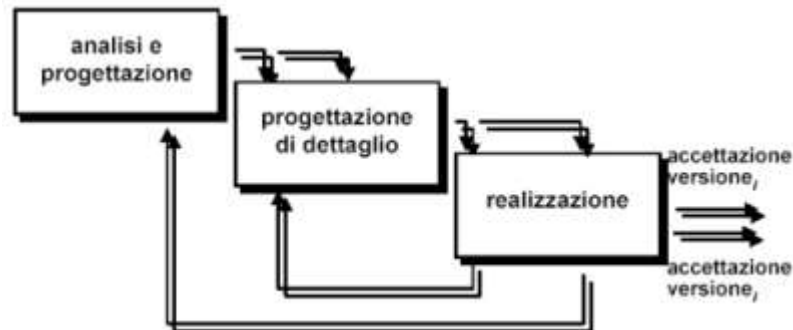
Naturalmente questo proto-modello scade nei modelli già studiati (waterfall, incremental, evolutionary) quando ci sono determinate condizioni. E' altresì vero che anche questo modello soffre di problemi legati alla sua natura intrinseca, primo su tutti il fatto che va bene per progetti interni a grandi aziende, ma difficilmente si adatta a realtà di produzione di software commerciale o a piccole entità (che in molti casi non avrebbe il tempo di rispettare tutti i passi e di investire tutte le risorse richieste).

## 2.3 - CICLI DI VITA IN ISO/IEC 12207

Qui di seguito presentiamo tre grafici per i cicli di vita come vengono rappresentati dallo standard ISO/IEC 12207; lo standard è necessario come strumento di modellazione che permetta di definire processi di sviluppo aziendali e permetta di adattarli e specializzarli per progetti particolari. Si possono quindi modellare i cicli di vita

classici (cascata, incrementale, evolutivo) utilizzando come componenti le attività dell' ISO/IEC 12207, e facendo in modo che i risultati di un'attività vengano usati come ingressi per la successiva.

## 2.4 - UNIFIED PROCESS



Questo modello è stato proposto nel 1999 da Grady Booch, Ivar Jacobson e James Rumbaugh.

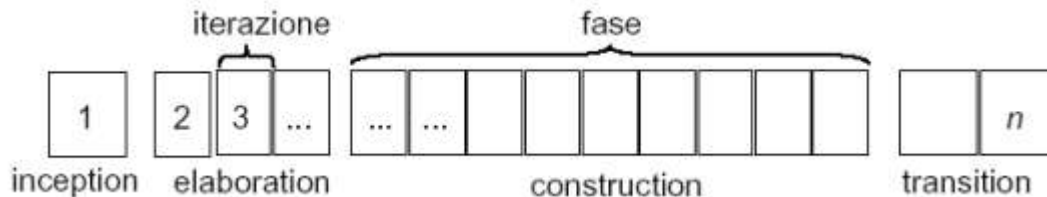
Lo Unified Process è un modello **risk driven** (come il modello a spirale), è **iterativo** (come l'incrementale), in quanto gli obiettivi sono piccoli e le scadenze sono chiare e, ad ogni iterazione, c'è rilascio del software; è agile (light), in quanto tutti gli elaborati sono opzionali, ed è adattativo (adaptive), in quanto i requisiti ed il progetto sono continuamente raffinati.

Il modello Unified Process è diviso in **settori**, **fasi** ed **iterazioni**.

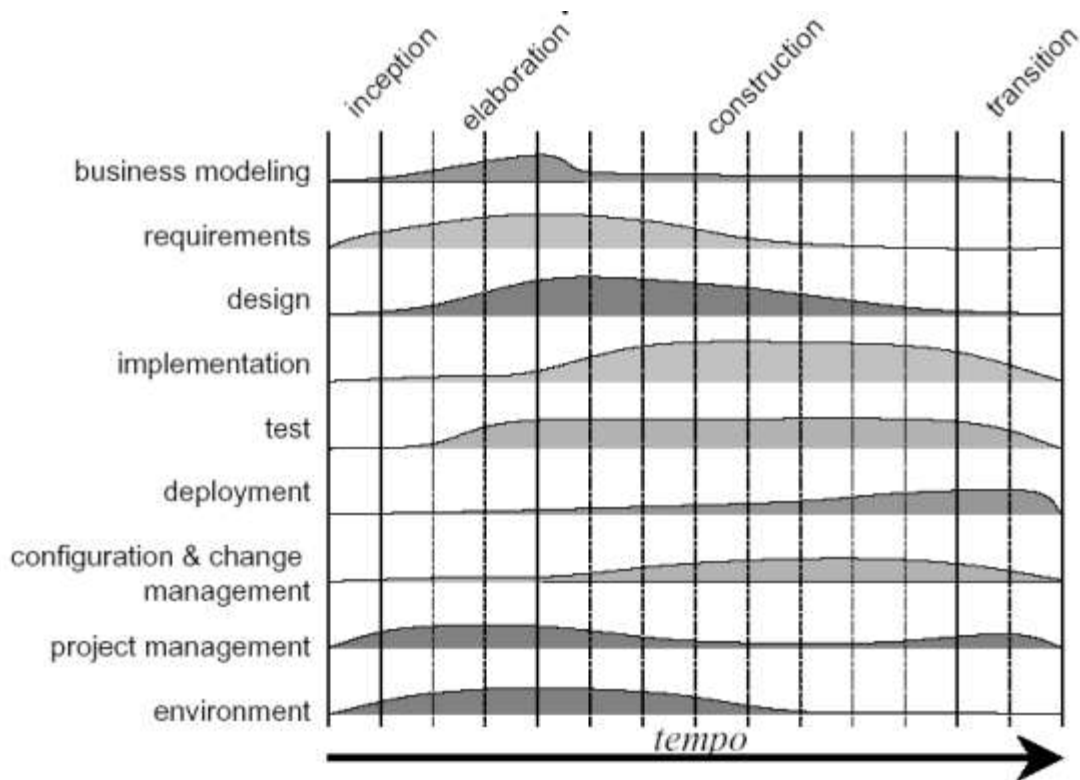
I settori sono:

- modello business;
- requisiti;
- progettazione;
- implementazione;
- test;
- deployment;
- gestione configurazioni e cambiamenti;
- gestione del progetto;
- ambiente.

Per ogni settore abbiamo la divisione in "fasi", che sono:



- **Avvio (Inception)** - In cui bisogna capire se il progetto è realizzabile con i vincoli di budget e di tempo e se esistono soluzioni alternative (make or buy). Si identificano i rischi e si sviluppa un eventuale prototipo;
- **Elaborazione (Elaboration)** - In cui bisogna ridurre al minimo i rischi di insuccesso; si fa l'analisi dei requisiti, l'analisi dei rischi, si sviluppa un'architettura base e si fa un piano per la fase di costruzione;
- **Costruzione (Construction)** - In questa fase si stabilizza il progetto (analisi) e l'architettura (disegno), si fa l'implementazione e il testing;
- **Transizione (Transition)** - In questa fase si mette in produzione il software (per la distribuzione presso il cliente), si fa il beta testing, si aggiustano le prestazioni, si crea documentazione aggiuntiva, si fa attività di formazione del personale, si creano tool per la migrazione dati da vecchi sistemi, si crea un kit per la vendita.



Ogni fase è divisa in iterazioni. Ciascuna iterazione ha degli obiettivi (ridimensionabili) e una scadenza inderogabile (timeboxed iterations). Le iterazioni sono brevi, circa 2 settimane. Alla fine di ciascuna iterazione si fa il punto della situazione e si programmano i compiti e la durata dell'iterazione successiva.

Lo strumento di supporto per lo Unified Process è il **development case**, un documento che mostra tutti gli elaborati e la loro evoluzione nel tempo.

### 2.3 - EXTREME PROGRAMMING

La metodologia eXtreme Programming (XP) è nata con il Manifesto di Snowbird (Utah), nel febbraio 2001. L'eXtreme Programming è una metodologia estremamente sensibile agli "aspetti sociali", come la comunicazione, il lavoro di gruppo, la soddisfazione personale, le relazioni interpersonali, qualità del lavoro, l'ambiente di lavoro. E' un modello **risk driven**, è agile, in quanto non bisogna produrre nessun documento, ed è relativamente semplice da imparare, ci sono infatti 4 valori, 5 principi e 12 pratiche da imparare. E' adatto a progetti con meno di 50 persone, e presuppone un forte coinvolgimento con il cliente.

#### VALORI

- **Comunicazione** - Discussioni libere tra tutte le persone coinvolte (programmatore, manager, utenti, clienti ecc.);
- **Semplicità** - Tutto ciò che non serve ora non si fa;
- **Feedback** - Per tutto ciò che si fa ci deve essere un feedback di correttezza e qualità;
- **Coraggio** - ... di fare la cosa che è chiaramente migliore anche se richiede molta fatica.

#### PRINCIPI

- **Feedback rapido** - Se il feedback non è rapido non serve;
- **Soluzioni semplici** - Inutile perdere tempo per cose che forse non servono;
- **Cambiamenti incrementali** - Niente stravolgimenti nell'architettura e nel codice. Grandi cambiamenti si fanno a piccoli passi (refactoring);
- **Adottare il cambiamento** - XP rende i cambiamenti poco onerosi e poco rischiosi quindi non si deve avere paura di cambiare l'architettura e il codice;
- **Lavoro di qualità** - Fare i test e molto altro; leggere "Lo Zen e l'arte della manutenzione della motocicletta".

## PRATICHE

- Planning game - Per ciascuna iterazione (breve) si pianificano i task;
- Small releases - Ciascuna iterazione da luogo ad un rilascio;
- Simple design - Architettura senza inutili fronzoli;
- Test-first programming - Procedure di testing automatico. Prima si scrive il codice di test e poi si sviluppa;
- Continuous integration - Si fa un'operazione di integrazione del software continua;
- **Refactoring** - Metodologia per cambiare il codice in maniera incrementale;
- Pair programming - Bisogna sempre seguire i passi dei requisiti, analisi, progetto, sviluppo, debugging e test;
- Collective ownership - Tutti gli sviluppatori possono modificare qualsiasi parte del codice;
- 40-hour week - Niente straordinari;
- On-site customer - L'utente/cliente deve essere consultato di frequente;
- Methaphor - Sviluppare un linguaggio comune con gli utenti/clienti;
- Coding standard - Poichè il codice deve essere modificato da tutti, deve essere comprensibile a tutti.

Nella metodologia XP, si hanno una serie di passi principali:

- Pianificazione flessibile (coinvolge utenti e programmatori);
- Rilasci frequenti (due/quattro settimane);
- Utilizzo di metafore condivise;
- Progetti semplici (comprensibili a tutti e refactoring);
- Verifica (testing di unità e sistema);
- Programmazione a coppie (con un singolo terminale);
- Collettivizzazione del codice (accesso libero a tutti, sapere comune).

## APPROFONDIMENTO: REFACTORING

*(tratto dagli appunti del prof. Crescenzi, Università di Roma, [www.dia.uniroma3.it/~crescenzi](http://www.dia.uniroma3.it/~crescenzi))*

Il refactoring è **il processo di modificare un sistema software senza alterarne il comportamento esterno pur migliorandone la struttura**, è anche un modo disciplinato di ripulire il codice esistente minimizzando le possibilità di introdurre nuovi bugs.

Alcune scuole di pensiero (come XP) hanno rivalutato la concezione secondo la quale il codice va scritto dopo la fase di progettazione, anticipandone, di fatto, la scrittura; inoltre il refactoring può intervenire quando l'integrità del sistema, e la qualità del codice, svanisce a causa di modifiche progressive.

Talvolta un sistema degenera al punto che ogni modifica diventa proibitiva, il refactoring è una tecnica per fronteggiare questo scadimento del codice rispetto ai requisiti, attuali e futuri.

Il processo di refactoring è composto da passi minimali di modifica del codice, ma ciascuna modifica è talmente semplice da risultare agevole e prova di inconvenienti (non deve introdurre nuovi bug); l'effetto cumulativo di piccole modifiche può migliorare drasticamente il progetto.

Con il termine **Refactoring** si identifica:

- Il singolo passo di ristrutturazione del codice per migliorarne la qualità senza alterarne il comportamento osservabile;
- Il processo di ristrutturazione del codice, cumulando tanti piccoli passi di ristrutturazione senza alterare il comportamento osservabile.

In definitiva l'obiettivo del refactoring è quello di rendere il codice più semplice da "maneggiare" e da comprendere.

## Perchè?

- Per migliorare la progettazione del software, contrastando il "Software Decay" (decadimento del software), che si presenta quando gli obiettivi di breve termine contrastano con quelli di medio-lungo termine;
- Per rendere il codice più comprensibile, pensando allo sviluppatore futuro del codice (soprattutto se quello sei sempre te);
- Per trovare e correggere bugs;

- Per programmare velocemente e con qualità.

### Come?

- Un prerequisito fondamentale è la disponibilità di solidi test che minimizzino la probabilità di introdurre bugs, consentano di "misurare" l'affidabilità del codice e contrastino la "paura" di fare cambiamenti;
- Risulta fondamentale mantenere il controllo del codice che non deve mai *allontanarsi troppo dal funzionare*;
- Per ogni singolo passo si costruiscono i test per il risultato del refactoring, si ristrutturano il codice, e si eseguono test e si fanno correzioni fino a ripristinare il completo funzionamento del codice;
- Il codice deve tornare a funzionare dopo ogni singolo passo di refactoring.

### Quando?

- Continuamente, con piccole sessioni unicamente dedicate al refactoring;
- In genere l'esigenza del refactoring scaturisce quando si vuole fare qualcos'altro (aggiungere una funzionalità, correggere un bug), ed una sessione di refactoring aiuta in tal senso.

E' da notare che **il target dei refactoring spesso sono Design Pattern.**

---

## LEZIONE 3 - ANALISI DEI REQUISITI

### 3.0 - INTRODUZIONE

Per lo sviluppo di un software bisogna fare un'attività di analisi, e quindi

- Studiare e definire il problema da risolvere
  - Per identificare il prodotto da commissionare;
  - Per capire cosa deve essere realizzato;
  - Per definire completamente gli accordi committente/fornitore.
- Valutare le implicazioni economiche e sulla qualità del prodotto
  - La soddisfazione del committente è relativa ai requisiti;
  - I requisiti possono essere sia espliciti che impliciti.

Secondo lo Standish Group, in un documento del 1995 (disponibile in formato HTML, [QUI](#)), le dieci maggiori cause di abbandono di un processo software sono:

1. **Requisiti incompleti;**
2. **Scarso coinvolgimento degli utenti;**
3. **Mancanza di risorse;**
4. **Attese irrealistiche;**
5. Mancanza di supporto esecutivo;
6. **Modifiche a specifiche e requisiti;**
7. Mancanza di Pianificazione;
8. Non c'era più bisogno;
9. Mancanza di Gestione IT;
10. **Ignoranza tecnologica.**

Per evitare questi problemi è quindi importante comprendere il "dominio", studiando il prodotto e il suo campo di applicazione, infatti solo conoscendo bene il problema si può risolverlo. Bisogna quindi acquisire le competenze tramite la documentazione esistente, la conduzione di interviste e tramite lo studio di soluzioni esistenti.

### 3.1 - L'ANALISI TRADIZIONALE

Per fare questo l'analisi tradizionale opera con i seguenti passi:

- Studio di fattibilità;
- Analisi dei requisiti - Basata sull'uso del linguaggio naturale (dominio, glossario, requisiti);
- Specifica - Basata sull'uso di linguaggi formali o semi-formali (funzioni, profilo operativo);
- Progettazione Top-Down e Realizzazione.

### 3.2 - L'ANALISI MODERNA

L'analisi moderna, quella basata su linguaggi object oriented, consiste dei seguenti passi:

- Studio di fattibilità;
- Analisi OO - Basata su formalismi grafici (dominio, glossario, requisiti).  
Collegata alla progettazione tramite l'identificazione delle classi;
- Progettazione OO - Utilizzo di componenti prefabbricati o realizzazione di componenti riusabili;
- Programmazione ad oggetti.

### 3.3 - STUDIO DI FATTIBILITÀ

Lo studio di fattibilità cerca di identificare i **rischi, costi e benefici** di una realizzazione; lo studio di fattibilità viene fatto per capire le prospettive del committente e del fornitore, ed è uno studio basato su dati eterogenei e spesso incerti, che però portano alla definizione e valutazione di possibili scenari.

Grazie allo studio di fattibilità si valuta e si decide se procedere o meno allo sviluppo del software, il costo di tale studio deve essere stabilito e non deve essere fatta alcuna attività di ricerca.

#### Obiettivi dello studio di fattibilità

- **Fattibilità tecnica e organizzativa**
  - Soluzioni algoritmiche e architettoniche;
  - Hardware per il supporto a tempo di esecuzione;

- Strumenti per la realizzazione.
- **Individuazione dei rischi**
  - Complessità e incertezza.
- **Analisi di costi e benefici**
  - Confronto tra il mercato attuale e quello futuro;
  - Costo della produzione e redditività dell'investimento
- **Scadenze temporali**

Uno dei punti essenziali dello studio di fattibilità è l'**esame delle alternative**, come la alternative architettoniche (sistema centralizzato o distribuito), o sulle modalità di realizzazione ("make or buy", avvio, esercizio e manutenzione del sistema, e la formazione e assistenza agli utenti).

### 3.4 - ANALISI DEI REQUISITI

#### Definizione di requisito

1. *Una condizione o capacità necessaria a un utente per risolvere un problema;*
2. *Una condizione (capacità) che deve essere soddisfatta (posseduta) da un sistema per soddisfare un contratto.*

#### Categorie dei requisiti

- Requisiti funzionali
  - Tradizionalmente i requisiti a cui è stato dato maggior valore;
  - Il prodotto è visto come un'insieme di funzionalità;
  - Quando necessario devono essere espressi formalmente.
- Caratteristiche di qualità
  - Una visione più ampia e moderna;
  - Efficienza, affidabilità, usabilità...;
  - Modelli per la qualità del software;
  - Norme per identificare le caratteristiche di qualità.

#### Glossario

- Definizione dei termini chiave del dominio
  - Proprietà di chiusura (tutti);
  - Proprietà di sinteticità (e soli);
  - Verificato e approvato dal committente.
- Interviste e glossario
  - Strumento per la conduzione di interviste;
  - Interviste per verificare il glossario.

#### 3.4.1 - ACQUISIZIONE

- Interviste
  - Strutturate
  - Non strutturate
- Questionari scritti
  - Scelte multiple, ...
- Osservazione dei futuri utenti al lavoro
- Studio di documenti
- Produzione di prototipi
  - Evolutivi
  - Usa e gett

#### Proprietà del documento dei requisiti

- Completo
- Ben organizzato
- Privo di inconsistenze
- Privo di ambiguità

- Privo di ridondanze
- Privo di imprecisioni terminologiche
- Privo di dettagli tecnici

#### **3.4.2 - VALIDAZIONE**

- Eseguita su un documento già organizzato
- Walkthrough
- Ispezione
  - Lettura "strutturata" dei documenti
  - Esempio: tecnica del lemmario
  - Efficacia provata sperimentalmente (60% dei problemi)
- Matrice delle dipendenze

#### **3.4.3 - NEGOZIAZIONE**

- Modifiche alla lista di requisiti, divisione in classi
- Requisiti obbligatori
  - Irrinunciabili per il cliente
- Requisiti desiderabili
  - Non necessari, ma utili
- Requisiti opzionali
  - Relativamente utili, oppure contrattabili in seguito

#### **3.4.4 - GESTIONE**

- Identificazione, classificazione
  - Identificatore unico (DB. forniscono utili funzionalità)
  - Numero sequenziale basato sulla struttura del documento (2.4.7)
  - Coppia <CATEGORIA, NUMERO>
- Gestione di cambiamenti
  - Valutazione della fattibilità tecnica e impatto sul resto del progetto
- Tracciabilità
  - Requisiti vs. elementi specifica vs. componenti del sistema
  - Strumenti CASE

### **3.5 - SEMINARIO: LA MISSIONE GPL**

Visualizza il caso di studio della missione GPL (formato PDF) - [QUI](#)

---

# LEZIONE 10 (e 11) - DESIGN PATTERNS

## 10.1 - Progettare Software

- Il progetto del software basato su oggetti riusabili è complesso e difficile
- I progettisti sfruttano le proprie esperienze : quando sviluppano un nuovo sistema in genere non partono da zero, ma riusano soluzioni progettuali che hanno funzionato in passato

### Come si fa a comunicare l'esperienza progettuale?

- Attraverso i **Design Patterns**
- I design pattern sono emersi da alcuni anni come una delle tecniche piu' efficaci per trasferire la conoscenza e l'esperienza dei progettisti, rendendola disponibile a tutti in modo comprensibile e concreto.
- I pattern sono proprio un modo : "Di registrare le esperienze nella progettazione di software object-oriented"

Quindi i D.P. aiutano a :

- Costruire del software che sia riusabile
- Evitare scelte che compromettano il suo riutilizzo
- Inoltre possono migliorare la documentazione e la manutenzione di sistemi esistenti
- Aiutano a progettare "in modo giusto e in minor tempo"

## 10.2 - Cosa sono i Design Patterns ?

Definizione: Ogni pattern descrive un problema specifico che ricorre più volte e descrive il nucleo della soluzione a quel problema, in modo da poter utilizzare tale soluzione un milione di volte, senza mai farlo allo stesso modo.

In pratica un design pattern è una regola tripartita, che esprime una *relazione* tra un *contesto*, un *problema* ed una *soluzione*.

### 10.2.1 - Un Design Pattern

- Un design pattern *nomina*, *astrae* ed *identifica* gli aspetti chiave di una struttura di design comune, che la rendono utile per creare un design OO riusabile.
- Identifica
  - Le classi e le istanze che vi partecipano
  - I loro ruoli
  - Come collaborano
  - La distribuzione delle responsabilità
- Astrae una struttura di design ricorrente
- Descrive ed assegna nomi ai vari componenti
- Distilla l'esperienza di progettazione
- Ogni pattern si focalizza su un particolare problema di OO design:
  - Descrive quando la soluzione può essere applicata
  - Le conseguenze di tale applicazione
  - I "compromessi"
    - Vantaggi
    - Svantaggi
- Poiché i progetti devono essere implementati, un pattern deve anche fornire
  - Suggerimento per l'implementazione
  - Esempio di utilizzo
- I design pattern non sono solo teoria!

### 10.2.2 - Come sono fatti i Design Patterns

- Il **nome** del pattern, è utile per descrivere la sua funzionalità in una o due parole.
- Il **problema** nel quale il pattern è applicabile. Spiega il problema e il contesto, a volte descrive dei problemi specifici del design mentre a volte può descrivere strutture di classi e oggetti. Può anche

includere una lista di condizioni che devono essere soddisfatte precedentemente perché il pattern possa essere applicato.

- La **soluzione** che descrive in modo astratto come il pattern risolve il problema. Descrive gli elementi che compongono il design, le loro responsabilità e le collaborazioni.
- Le **conseguenze** portate dall'applicazione del pattern. Spesso sono trascurate ma sono importanti per poter valutare i costi-benefici dell'utilizzo del pattern.

### 10.2.3 - Descrizione DP

- Il formato che si sceglie per la descrizione di un pattern non è importante, purché la descrizione sia completa e accurata.
- La descrizione di un design pattern viene comunque data usando un formato preciso, articolato in diversi paragrafi

### 10.2.4 - Esempio Descrizione DP

- **Nome e classificazione** del pattern
- **Sinonimi**: altri nomi del pattern
- **Scopo**: cosa fa il pattern? a cosa serve?
- **Motivazione**: scenario che illustra un design problem
- **Applicabilità**: situazioni in cui si applica il pattern
- **Struttura**: rappresentazione delle classi in stile OMT
- **Partecipanti**: classi e oggetti inclusi nel pattern
- **Collaborazioni**: come i partecipanti collaborano
- **Conseguenze**: come consegue i suoi obiettivi il pattern?
- **Implementazione**: che tecniche di codifica sono necessarie?
- **Codice di esempio**: scritto in un linguaggio a oggetti
- **Usi noti**: esempi d'applicazione del pattern in sistemi reali
- **Pattern correlati**: con quali altri pattern si dovrebbe usare?

### 10.2.5 - Perché utilizzare i DP

- Il programma sarà
  - flessibile
  - Modulare
  - Riusabile
  - Comprensibile

### 10.2.6 - Obiettivi DP

- Creare una letteratura all'interno della comunità Object-Oriented
- Aiutare gli sviluppatori a risolvere problemi già trattati
- Creare un linguaggio per comunicare intuizioni ed esperienza su problemi e sulle loro soluzioni
- Mostrare che la soluzione descritta è:
  - *Utile* (ricorrenza)
  - *Utilizzabile* (contesto specifico)
  - *Usata* (assodata)

### 10.2.7 - Linguaggio di Patterns

#### Definizione:

*Un linguaggio di pattern, o sistema di pattern, o dizionario di progetto sw, è una collezione di pattern per architetture sw, che include le istruzioni per implementarli, combinarli e usarli nella prassi dello sviluppo sw.*

### 10.3 - Un Catalogo DP

- Un esempio di catalogo DP è presente nel libro "*Design Patterns: Elements of Reusable Object Oriented Software*".
- Il libro classifica 23 design pattern e li suddivide in 3 categorie: **Creational, Behavioral e Structural**.
- Alcuni pattern sono utilizzati assieme
- Alcuni sono alternative di altri
- Altri sono simili ma hanno intenti differenti

#### 10.3.1 - Categoria Creational

Design Patterns	Descrizione
Builder	Separa la costruzione di un oggetto complesso dalla sua rappresentazione in modo da poter usare lo stesso processo di costruzione per altre rappresentazioni
Abstract Factory	Provvede ad un'interfaccia per creare famiglie di oggetti in relazione senza specificare le loro classi concrete
Factory Method	Definisce un'interfaccia per creare un oggetto ma lascia decidere alle sottoclassi quale classe istanziare
Prototype	Specifica il tipo di oggetto da creare usando un'istanza prototipo e crea nuovi oggetti copiando questo prototipo
Singleton	Assicura che la classe abbia una sola istanza e provvede un modo di accesso

- Questa categoria raccoglie i pattern che forniscono un'astrazione per il processo di istanziazione.
- Questi pattern permettono di rendere un sistema indipendente da come sono creati, rappresentati e composti gli oggetti al suo interno.

### 10.3.2 - Categoria Structural

Design Patterns	Descrizione
Adapter	Converte l'interfaccia di una classe in un'altra permettendo a due classi di lavorare assieme anche se hanno interfacce diverse.
Bridge	Disaccoppia un'astrazione dalla sua implementazione in modo che possano variare in modo indipendente.
Composite	Compone oggetti in strutture ad albero per implementare delle composizioni ricorsive
Decorator	Aggiunge nuove responsabilità ad un oggetto in modo dinamico, è un alternativa alle sottoclassi per estendere le funzionalità
Facade	Provvede un'interfaccia unificata per le interfacce di un sottosistema in modo da rendere più facile il loro utilizzo
Proxy	Provvede un surrogato di un oggetto per controllarne gli accessi
Flyweight	Usa la condivisione per supportare in modo efficiente un gran numero di oggetti con fine granularità

- I pattern di questa categoria sono dedicati alla composizione di classi e oggetti per creare delle strutture più grandi.
- È possibile creare delle classi che ereditano da più classi per consentire di utilizzare proprietà di più superclassi indipendenti.
- Ad esempio permettono di far funzionare insieme delle librerie indipendenti.

### 10.3.3 - Categoria Behavioral

Design Patterns	Descrizione
Chain of Responsibility	Evita l'accoppiamento di chi manda una richiesta con chi la riceve dando a più oggetti la possibilità di maneggiare la richiesta.
Command	Incapsula una richiesta in un oggetto in modo da poter eseguire operazioni che non si potrebbero eseguire.
Interpreter	Dato un linguaggio, definisce una rappresentazione per la sua grammatica ed un interprete per le frasi del linguaggio.
Iterator	Provvede un modo di accesso agli elementi di un oggetto aggregato in modo sequenziale senza esporre la sua rappresentazione sottostante
Mediator	Definisce un oggetto che incapsula il modo in cui un insieme di oggetti interagisce in modo da permettere la loro indipendenza
Memento	Cattura e porta all'esterno lo stato interno di un oggetto senza violare l'incapsulazione in modo da ripristinare il suo stato più tardi

Design Patterns	Descrizione
Observer	Definisce una dipendenza 1:N tra oggetti in modo che se uno cambia stato gli altri siano aggiornati automaticamente
State	Permette ad un oggetto di cambiare il proprio comportamento a seconda del suo stato interno, come se cambiasse classe di appartenenza
Strategy	Definisce una famiglia di algoritmi, li incapsula ognuno e li rende intercambiabili in modo da cambiare in modo indipendente dagli utilizzatori
Template method	Definisce lo scheletro di un algoritmo in un'operazione lasciando definire alcuni passi alle sottoclassi
Visitor	Rappresenta un'operazione da fare sugli elementi della struttura di un oggetto. Lascia definire nuove operazioni senza cambiare classe degli elementi

- Questi pattern sono dedicati all'assegnamento di responsabilità tra gli oggetti e alla creazione di algoritmi.
  - Una caratteristica comune in questi pattern è il supporto per seguire le comunicazioni che avvengono tra le classi.
  - l'utilizzo di questi pattern permette di dedicarsi principalmente alle connessioni tra oggetti lasciando in disparte la gestione dei flussi di controllo.
-

## LEZIONE 14 - IL PROCESSO SOFTWARE

### 14.0 - INTRODUZIONE

Come abbiamo detto l'ingegneria del software non è solamente una materia in cui si pensa a come progettare software, ma si cerca di studiare anche la fase organizzativa dell'azienda nell'ottica di una migliore gestione della propria forza lavoro e un incremento della produttività, attraverso lo studio dei processi aziendali. Per fare ciò ci viene in aiuto lo standard ISO 12207.

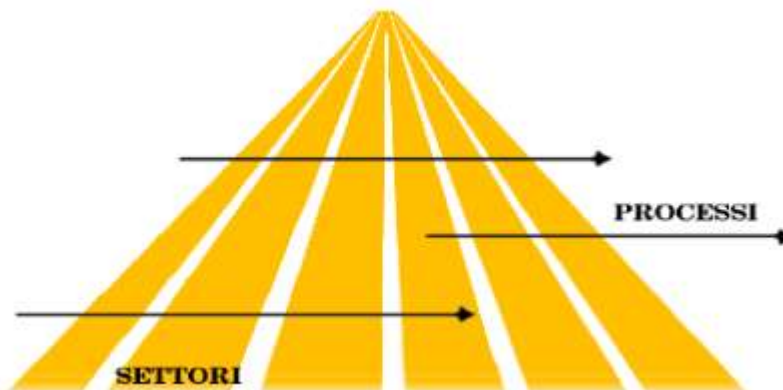
Il processo, secondo una definizione dell'ISO 9000, è:

***Un insieme di attività correlate  
che trasformano ingressi in uscite***



Il processo standard è il processo di base usato per stabilire un modo comune di svolgere funzioni aziendali; da questo si passa al processo definito, che è una specializzazione del processo standard, necessaria per adattarlo a esigenze specifiche; si arriva quindi al progetto che è un'istanziamento di un processo definito che utilizza risorse aziendali per raggiungere obiettivi stabiliti.

Mostriamo un semplice grafico di un'azienda la cui organizzazione è basata sull'identificazione dei suoi processi:



All'interno dell'azienda esistono varie classi di processi; ci sono, ad esempio, quelli legati direttamente alla missione e quelli che realizzano l'infrastruttura; tanto per fare degli esempi, si può pensare allo sviluppo di prodotti software a pacchetti, sviluppo di sistemi software su richiesta, servizi di assistenza e manutenzione, ricerche di mercato, ricerca tecnologica e formazione, e l'amministrazione.

### 14.1 - ISO/IEC 12207

L'ISO/IEC 12207 contiene una descrizione approfondita dei processi del ciclo di vita del software, ed è infatti il

modello più noto e riferito, anche se ne esistono altri. Questo modello è ad alt livello, ed identifica i processi dello sviluppo software, ed ha una struttura modulare che permette, nel processo di specializzazione, di identificare le entità responsabili dei processi ed i prodotti dei processi.

Secondo questo modello si hanno processi (processes), che sono divisi in attività (activities) che, a loro volta, sono divisi in compiti (tasks). Così si ha una struttura modulare (perchè i processi si interfacciano), ma con una forte coesione (perchè i compiti sono chiusi). ISO 12207 descrive cinque processi primari, otto processi di supporto e quattro processi organizzativi, per un totale di diciassette processi che hanno lo scopo di eliminare tutti gli sprechi di tempo e risorse, eliminando le particolari attività per un progetto specifico.

#### **14.1.1 - PROCESSI PRIMARI**

- Acquisizione (gestione dei sottofornitori);
- Fornitura (rapporti con il cliente);
- Sviluppo di sistemi software (Development Process);
- Gestione operativa (installazione e servizi);
- Manutenzione.

#### **14.1.2 - PROCESSI DI SUPPORTO**

- Documentazione del progetto;
- Gestione delle versioni e delle configurazioni;
- Assicurazione della qualità;
- Verifica;
- Validazione;
- Revisioni congiunte con il cliente;
- Verifiche ispettive interne;
- Risoluzione dei problemi.

#### **14.1.3 - PROCESSI ORGANIZZATIVI**

- Gestione dei progetti;
- Gestione delle infrastrutture;
- Miglioramento del processo;
- Formazione del personale.

### **14.2 - ATTIVITA' E COMPITI NEL PROCESSO DI SVILUPPO (ORIGINALE)**

Visualizza il documento dell'ISO/IEC 12207 (formato PDF) - [QUI](#)

### **14.3 - SPECIALIZZAZIONE**

Si può prendere un particolare insieme di processi e specializzarlo in base alle aziende, infatti ci sono un insieme di elementi ben definiti, indipendenti dal ciclo di vita scelto, dalle tecnologie, dal settore applicativo e dalla documentazione.

Oppure possiamo specializzarlo per progetto, pianificando la specializzazione, e poi specializzando ogni processo grazie alla definizione dello scenario, delle eventuali attività e compiti aggiuntivi e all'organizzazione dei processi; a questo punto si conduce il progetto pilota ed eventualmente si formalizza ed istituziona quanto ottenuto.

I fattori di specializzazione che possono intervenire sono la dimensione del progetto, la sua complessità, i suoi rischi, la competenza ed esperienza delle risorse ed i fattori dipendenti dal contratto.

### **14.4 - EVOLUZIONE DEGLI STANDARD**

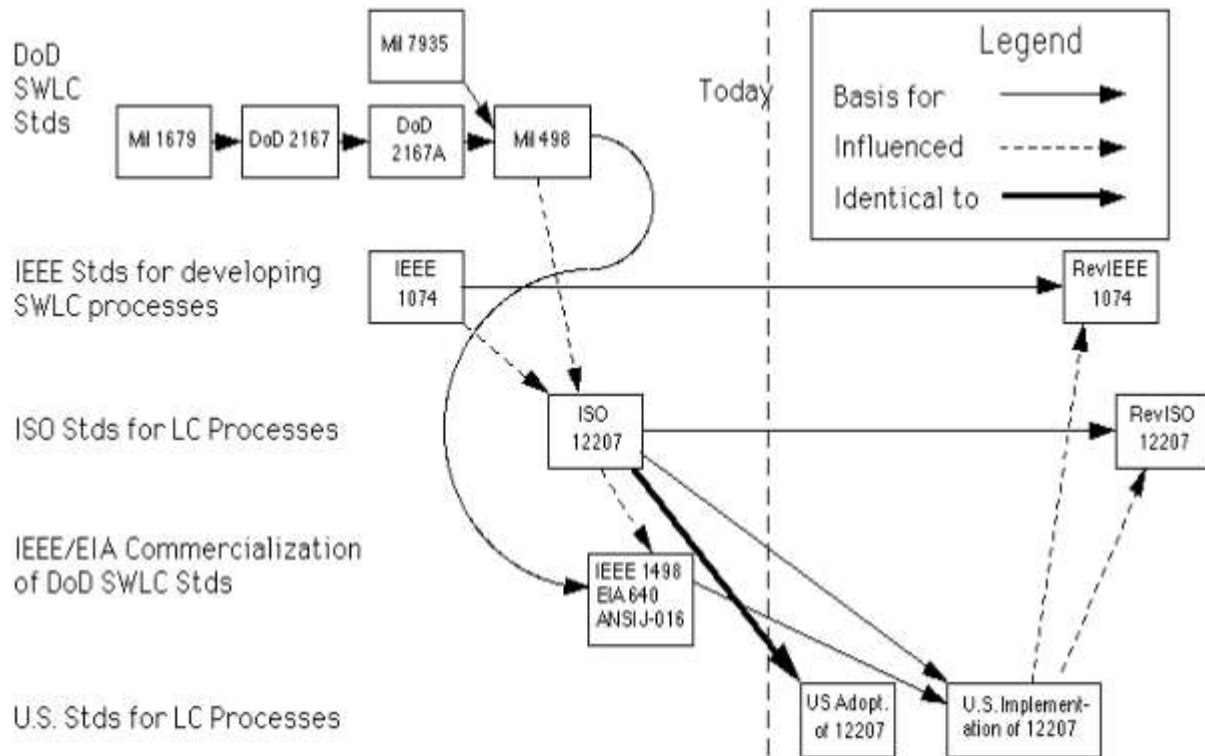
Creare uno standard è un'iniziativa che, generalmente, nasce dal committente per facilitare le attività di controllo, collaudo ed accettazione di un sistema.

Le prime applicazioni si hanno in america durante la II Guerra Mondiale, risale al 1950, infatti, il DoD Mil105A, che è una razionalizzazione delle esperienze belliche, incentrata sul processo di collaudo ed accettazione, misurabili tramite prove (Acceptable Quality Level).

Negli anni sono nati molti standard, e si sono contaminati a vicenda; come mostrato in figura, il Mil 2167 per sistemi critici (1988) ed il Mil 7935 per i sistemi informativi (1988) hanno dato vita al Mil 498 (1994); contemporaneamente si è avuta la nascita dell'IEEE 1074 e dell'IEEE 1498 (1992); tutti insieme hanno dato vita

all'ISO/IEC 12207. - Vedere la figura sottostante -

## Evolution of Life-Cycle Standards



Volendo continuare a parlare di standard, si può dire che ci sono anche gli standard ESA, che sono applicati dall'ESA stessa e dai suoi fornitori, esistono il PSS-05 Software Engineering Standards, del 1991, che è molto vicino all'IEEE 1074, ma è più uno standard per i prodotti e le procedure; poi abbiamo l'ECSS-Q-80A, del 1996, che assicura la qualità del software, e l'ECSS-E-40A, del 1999, che regola la produzione di software.

Le tendenze sono di usare lo standard come modello da applicare, per definire le procedure (ESA) e per definire i processi e le attività da specializzare (12207). Ma lo standard può essere usato anche modello di valutazione, usando modelli più generali per coprire contesti diversi o per identificare le "best practices" (pratiche migliori); citiamo a titolo informativo il CMM, SPICE e l'ISO/IEC TR 15504.

# LEZIONE 17 - MISURAZIONE DEL SOFTWARE

*Quando puoi misurare quello di cui stai parlando, e puoi esprimerlo con dei numeri, lo conosci abbastanza; ma quando non puoi misurarlo, quando non puoi esprimerlo con dei numeri, la tua conoscenza è scarsa ed insoddisfacente.*

Lord Kelvin, 1889

## 17.0 - INTRODUZIONE

In qualsiasi disciplina scientifica è importante poter misurare, infatti la finalità di una misura è di rendere oggettivi i risultati di eventuali esperimenti, in modo da poter rendere un determinato processo ripetibile e confrontabile; inoltre da una serie di misure si possono ricavare modelli matematici utili per la nostra disciplina. Oggettivamente una misura ha dei limiti, nel senso che può non essere esatta (realtà fisiche) o può essere un'astrazione (stime).

Possiamo quindi definire

- La **MISURAZIONE**, il processo che assegna numeri o simboli ad attributi di entità fisiche, secondo regole definite;
- La **MISURA**, il risultato della misurazione;
- La **METRICA**, un insieme di regole per definire le entità da misurare (es. persona), gli attributi rilevanti e le unità di misura (es. età), le procedure per assegnare numeri e simboli (es. 18 anni).

Secondo l'ISO/IEC 9126, le **caratteristiche** classificano gli aspetti ritenuti essenziali (e non sono misurabili), gli **attributi** (o sottocaratteristiche) rappresentano una classificazione più fine delle caratteristiche (non misurabili), e gli **indicatori** (o criteri ingegneristici) sono proprietà misurabili degli attributi. Gli indicatori vengono usati per misurare caratteristiche difficilmente misurabili, infatti le metriche per stimare caratteristiche permettono di identificare gli attributi misurabili e di stimare (con incertezza) le caratteristiche non misurabili; ad esempio la dimensione è un attributo misurabile, mentre il costo di manutenzione è una caratteristica stimata.

### 17.0.1 - Le metriche nella produzione

Servono, nel processo di produzione, una serie di strumenti di valutazione e controllo. Infatti si possono misurare:

- **PROCESSI** - Procedure astratte;
- **PROGETTI** - Attività concrete legate a tempo e risorse;
- **PRODOTTI** - Beni e servizi in uscita dai progetti;
- **RISORSE** - Elementi necessari per istanziare progetti.

Inoltre esistono due tipi di attributi per le metriche:

- *Interni*, misurabili in termini delle entità;
- *Esterni*, misurabili in termini dell'ambiente.

### 17.0.2 - Le metriche nel software

L'uso di metriche per il software è un problema aperto, questo perchè un software è difficile da misurare, a causa dei diversi aspetti che vengono presentati, dall'ambiente e dalle tecnologie che permettono di creare un software. Le metriche per il software sono concepite basandosi sul codice, sulle funzionalità o sul suo comportamento.

## 17.1 - LINEE DI CODICE

La base della metrica LoC (Line of Code - Linee di Codice) è che la lunghezza del programma può essere usata per prevedere le caratteristiche di un programma come lo sforzo e la facilità di manutenzione. La metrica LoC viene usata per misurare la dimensione del software; questo tipo di metrica, molto intuitiva, è anche tra le più usate.

### VANTAGGI

1. Facile da misurare.

### LIMITI

1. Viene definita sul codice. E' impossibile, per esempio, misurare la dimensione delle specifiche;
2. Caratterizza solamente un singolo aspetto della dimensione, cioè la lunghezza, non prendendo in considerazione funzionalità o complessità;
3. Una cattiva progettazione del software può causare un numero eccessivo di linee di codice;
4. Dipende dal linguaggio;
5. Gli utenti difficilmente la comprendono.

### 17.1.1 DSI e COCOMO (CONstructive COst MOdel)

Le DSI sono le "Delivered Source Instructions", basate sulle LoC e definite come segue:

- Soltanto le linee di codice sorgente che verranno incluse nel prodotto finale vengono incluse; programmi di test, debug e di supporto sono escluse;
- Le linee di codice sorgente vengono create dai programmatori; codice creato automaticamente è escluso;
- Una istruzione è una linea di codice;
- Le dichiarazioni vengono considerate istruzioni;
- I commenti vengono considerati istruzioni.

La tecnica COCOMO viene usata per calcolare lo "sforzo" di un progetto software. Il COCOMO è formato da tre modelli, quello base (basic), applicabile quando si conosce poco del progetto, quello intermedio (intermediate), applicato dopo che i requisiti sono specificati, e quello avanzato (advanced), che viene applicato quando la progettazione è completata. La formula generale del COCOMO è:

$$E = aS^bF$$

Dove E è lo sforzo, S è la dimensione misurata in migliaia di istruzioni (K-DSI), F è un valore di aggiustamento, a e b sono valori predefiniti secondo la complessità del modello (organic, semi-detached, embedded).

E' possibile calcolare anche la durata (D - Duration) di un progetto, conoscendo il coefficiente P, ovvero lo sforzo per persone/mese, ed usando la seguente formula,

$$D = aP^b$$

dove a e b sono dipendenti da una tabella apposita.

## 17.2 - SOFTWARE SCIENCE

Il metodo "software science" sviluppato da M.H.Halstead nel 1977, nasce principalmente per stimare lo sforzo di programmazione.

Le proprietà misurabili e contabili sono:

- n1 - numero di operatori unici o distinti che appaiono nell'implementazione;
- n2 - numero di operandi unici o distinti che appaiono nell'implementazione;
- N1 - uso totale di tutti gli operatori che appaiono nell'implementazione;
- N2 - uso totale di tutti gli operandi che appaiono nell'implementazione.

Da queste metriche Halstead definisce:

- I. Il vocabolario **n** come  $n = n1 + n2$ ;
- II. La lunghezza di implementazione **N** come  $N = N1 + N2$ .

Gli operatori possono essere "+" e "\*", ma anche gli indici "[...]" o il separatore di istruzioni ";".

Il numero di operandi consiste nel numero di espressioni letterali, costanti e variabili.

### 17.2.1 - EQUAZIONE DI LUNGHEZZA

La lunghezza del programma può essere stimata usando l'equazione:

$$N' = n1 \log_2 n1 + n2 \log_2 n2$$

### 17.2.2 - QUANTIFICAZIONE DELL'INTELLIGENZA

I medesimi algoritmi necessitano di maggiore considerazione in un linguaggio di programmazione a basso livello. E' facile programmare in Pascal piuttosto che in assembly. Cerchiamo di determinare ciò in un programma. Innanzitutto bisogna definire il

**Volume del programma:** Questa metrica è per la dimensione di ogni implementazione di ogni algoritmo.

$$V = N \log_2 n$$

**Livello del programma:** E' la relazione tra il volume del programma ed il volume potenziale. Soltanto l'algoritmo più chiaro ha questo valore uguale ad uno.

$$L = V^* / V$$

**Equazione del livello del programma:** E' un'approssimazione dell'equazione del livello del programma. Viene usata quando il valore del volume potenziale non è conosciuto a priori.

$$L' = n^*1n2 / n1N2$$

### Intelligenza

$$I = L' \times V = (2n2 / n1N2) \times (N1 + N2) \log_2(n1 + n2)$$

Questo valore risulta indipendente dal linguaggio di programmazione usato.

### 17.2.3 - SFORZO DI PROGRAMMAZIONE

Lo sforzo di programmazione è ristretto all'attività mentale richiesta per convertire un algoritmo esistente in una implementazione attuale in un linguaggio di programmazione. Si necessita delle seguenti metriche e formule:

**Volume Potenziale:** è una metrica usata per denotare i parametri corrispondenti in una formato algoritmico abbastanza piccolo.

$$V' = (n^*1 + n^*2) \log_2(n^*1 + n^*2)$$

### Equazione di "sforzo"

$$E = V/L = V2 / V'$$

Questo numero è la misura della difficoltà del programma.

**Equazione del tempo:** Un concetto sviluppato dallo psicologo John Stroud, è che definendo un "momento" il tempo richiesto dal cervello umano per fare una discriminazione elementare, si può definire un valore S, inteso come momenti per secondo (con S compreso tra 5 e 20) che, usato all'interno della seguente equazione, da il tempo necessario (stimato) per lo sviluppo di un programma.

$$T' = (n1N2(n1 \log_2 n1 + n2 \log_2 n2) \log_2 n) / 2n2S$$

### VANTAGGI

1. Non richiede un'analisi approfondita della struttura di programmazione;
2. Prevede la percentuale di errori;
3. Prevede lo sforzo di manutenzione;
4. Utile nella pianificazione e nei rapporti dei progetti;
5. Misura la qualità globale dei programmi;
6. Semplice da calcolare;
7. Può essere usata per ogni linguaggio di programmazione;
8. Numerosi studi industriali supportano l'uso di questa metodologia per prevedere lo sforzo di programmazione e il numero di bug di un programma.

### LIMITI

1. Dipende dal codice completo;
2. Non può essere usato come modello di valutazione preventivo. Generalmente per questa valutazione viene usato il modello di McCabe.

### 17.3 - COMPLESSITA CICLOMATICA

Il metodo per calcolare la complessità ciclomatica di un programma è stato inventato da McCabe nel 1976; la complessità ciclomatica viene definita come la complessità del flusso di controllo, ed è una funzione dei cammini **possibili ed indipendenti** sul grafo di flusso, inoltre fornisce una rappresentazione astratta del codice ed un valore numerico della complessità.

La complessità ciclomatica (V) di un grafo (G), ovvero il numero di percorsi indipendenti in G, può essere calcolata con la seguente formula:

$$V(G) = \text{Numero di archi (e)} - \text{Numero di nodi (n)} + 2 * \text{Numero delle componenti connesse (p)}$$

Semplificando  $V(G) = n^{\circ} \text{ di decisioni} + 1$

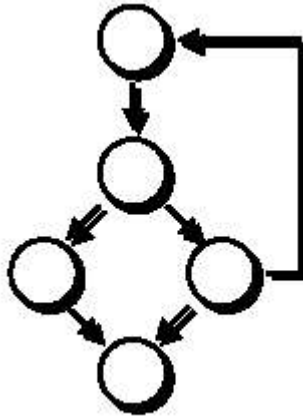
Il risultato di molteplici esperimenti suggerisce che i moduli hanno zero difetti quando la complessità ciclomatica di McCabe è circa  $7 \pm 2$ .

### 17.3.1 - ESEMPI

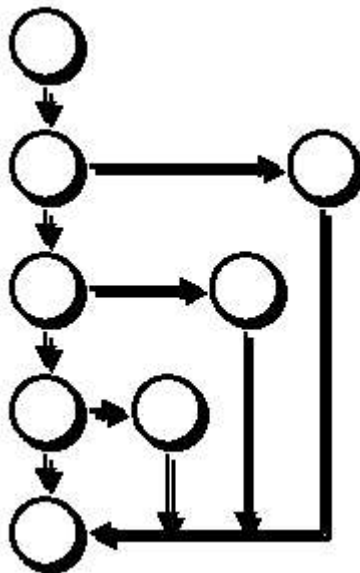
Sequenza -  $V(G) = 1$



Errore in difetto -  $V(G) = 3$  - Complessità reale maggiore



Errore in eccesso -  $V(G) = 4$  - Complessità reale inferiore



## VANTAGGI

1. Può essere usata come metrica di mantenimento;
2. Usata come metrica di qualità, fornisce la complessità di vari design;
3. Può essere calcolata nelle fasi iniziali del progetto (rispetto al metodo Software Science);
4. Misura lo sforzo minimo e la migliore area di concentrazione per i test;
5. Guida il processo di testing limitando la logica del programma durante lo sviluppo;
6. E' facile da applicare.

## LIMITI

1. Fallibilità dimostrata (ma usata);
2. Costosa da applicare prima di scrivere il codice;
3. Il calcolo della complessità riguarda la struttura e non i dati;
4. Cicli annidati e non-annidati hanno la stessa complessità; infatti strutture condizionali molto annidate sono difficili da comprendere.

## 17.4 - PUNTEGGIO FUNZIONALE

Il metodo dei "punti funzione" è stato ideato da Albrecht nel 1979 (presso la IBM), e serve per calcolare le metriche di produttività. Il metodo viene usato in due modi:

1. come una variabile di stima usata per sapere la "dimensione" di ogni elemento del software;
2. come metrica base collezionata da progetti precedenti ed usate in congiunzione con le variabili di stima per sviluppare proiezioni di costi e "sforzo".

L'approccio è quello di identificare e contare un numero di funzioni tipo uniche:

- Input Esterni (es. filename);
- Output Esterni (es. rapporti, messaggi);
- Interrogazione Esterne (input interattivi che necessitano di risposte);
- File di Interfaccia (file condivisi con altri sistemi software);
- File Logici Interni (invisibili fuori dal sistema).

Ognuna di queste funzioni viene valutata individualmente per la complessità e viene assegnato un valore pesato che varia da 3 a 15. Viene poi calcolata la somma di tutte le occorrenze (UFP), moltiplicando ogni conteggio delle funzioni con il valore pesato e, quindi, sommando tutti i valori. I pesi sono basati sulla complessità della funzione contata. Il metodo originale di Albrecht classificava i pesi come segue:

Function Type	Low	Average	High
External Input	x3	x4	x6
External Input	x4	x5	x7
Logical Internal File	x7	x10	x15
External Interface File	x5	x7	x10
External Inquiry	x3	x4	x6

Le decisioni possono essere determinate con questa tabella:

	1-5 Data element types	6-19 Data element types	20+ Data element types
0-1 File types referenced	Low	Low	Average
2-3 File types referenced	Low	Average	High
4+ File types referenced	Average	High	High

Per trovare il valore dei punti funzione (FP), UFP viene moltiplicato per il fattore di complessità tecnica, che può essere calcolato con la seguente formula:

$$TCF = 0.65 + (\text{somma dei fattori}) / 100$$

Ci sono 14 fattori di complessità tecnica. Ognuno di questi fattori viene valutato sulla base del suo grado di influenza, da quello meno influente al più influente:

1. Comunicazioni dei dati;
2. Elaborazione dati distribuita;
3. Prestazioni;
4. Carico sul sistema HW target;
5. Volume di transizioni;
6. Immissione dati on-line;
7. Facilità d'uso dell'applicazione;
8. Facilità di modifiche on-line;
9. Complessità di elaborazione;
10. Riusabilità;
11. Semplicità di installazione;
12. Semplicità di gestione operativa;
13. Installazioni plurime;
14. Semplicità di modifica.

Ognuna di queste caratteristiche può assumere un valore tra 0 (irrilevante) e 5 (essenziale). Quindi si ha che **FP = UFP x TCF = somma totale x (0.65 + 0.01 x sum(i=1->14) [V])**.

Si può notare che in genere la correzione oscilla sui valori di + o - il 35%.

### FP e Linee di Codice

Esiste una correlazione tra punti funzione (FP) e le linee di codice che, in un determinato linguaggio, sono necessarie per raggiungere un punto funzione. Si può notare che, più istruzioni ci vogliono, più il linguaggio è di "alto" livello; L'assembler, per esempio, per un punto funzione necessita di 320 istruzioni, ed è considerato di 1° livello. Mostriamo adesso una tabella con vari linguaggi/livelli/linee sulla quale si possono fare svariate considerazioni, come il fatto che solamente i linguaggi tra il livello 1 ed il livello 5 sono utilizzabili per applicazioni real-time, e rappresentano circa l'85% del software mondiale.

Linguaggio	Livello	Numero di istruzioni (SLOC) per punto funzione
Assembler	1	320
Macro Assembler	1,5	213
C	2	150
FORTRAN II		
BASIC Interpretato	2,5	128
ALGOL 68		
Fortran 77		
ANSI COBOL 85		
Jovial	3	105
ANSI COBOL 85		
Pascal		
BASIC Compilato	3,5	91
PL/I		
Modula 2		
RPG	4	80
Ada	4,5	71
Prolog		
LISP		
Forth		
BASIC ANSI	5	64
LOGO	5,5	58
English-based languages	6	53
Database languages	8	40
Decision support languages		
STRATAGEM	9	35
Statistical languages		
APL	10	32
Object oriented languages		
Objective C		
C++	12	27
SMALLTALK	15	21
DB Query languages	25	13
Spreadsheet	50	6
Linguaggi di 5ª generazione		
Graphic Icon Languages	75	4

#### VANTAGGI

1. Non è legato solo al codice;
2. E' indipendente dal linguaggio;
3. I dati necessari per il calcolo sono disponibili già nelle fasi iniziali del progetto. Si necessita solamente di

- una specifica dettagliata;  
4. Più accurato delle LoC stimate.

#### LIMITI

1. Conteggio soggettivo;
2. Difficile da automatizzare e complesso da calcolare;
3. Ignora la qualità dell'output;
4. Orientato a programmi gestionali tradizionali;
5. La previsione di sforzo usando i UFP a volte genera errori

#### 17.5 - PUNTI OGGETTO

Esiste una metrica chiamata di "Boehm et al." della fine degli anni 90 che utilizza dei punti oggetto. Purtroppo non è stato possibile trovare una spiegazione esauriente su internet.

#### 17.6 - STANDARD PER MISURE FUNZIONALI

Lo standard ISO/IEC 14143 (Functional Size Measurement) viene usato per:

- Definire la terminologia del settore;
- Definire i criteri per valutare le metriche funzionali;
- Definire i criteri per accreditare i professionisti che le usano.

I concetti che vengono definiti sono:

- Accuratezza di una misura funzionale;
- Accuratezza di una metrica funzionale;
- Ripetibilità e riproducibilità di una metrica funzionale;
- Soglia di sensibilità di una metrica funzionale;
- Applicabilità a un dominio funzionale.

#### 17.7 - METRICHE PER SOFTWARE OO

Per il software Object Oriented occorrono delle metriche dedicate, in quanto le metriche tradizionali non sono accurate, sorge il problema tra la complessità funzionale e quella strutturale (che non coincidono), non c'è linearità nel codice (quindi le LoC non funzionano) e l'uso di strutture e funzioni complesse come McCabe non funziona.

Bisogna quindi studiare delle:

- Metriche per i metodi;
- Metriche per le classi;
- Metriche per i sistemi.

Volendo si può usare McCabe come metrica di base.

#### Metrica per i metodi

$$MC = w_1MIC + w_2MLVC + w_3MCoC$$

MIC - Complessità dell'interfaccia

MLVC - Complessità delle variabili locali

MCoC - Complessità ciclomatica del codice

$w_i$  - Pesi determinati statisticamente

#### Metrica per le classi

$$CC = w_3CCL + w_4CCI$$

CCL =  $w_5ECCL + w_6ICCL$  - Complessità locale

ECCL =  $\sum(i)MIC_j$  - Complessità locale esterna

ICCL =  $w_7CACL + w_8CMCL$  - Complessità locale interna

CACL =  $\sum(h)CA_h$  - Complessità degli attributi locali

CMCL =  $\sum(k)MC_k$  - Complessità dei metodi locali

CA - c o CC della classe dell'attributo  
 $w_i$  - Pesi determinati statisticamente

### **Metrica per i sistemi**

Questa metrica considera solo la parte locale della classe e la complessità dell'uso delle classi negli attributi.

$$SC = \sum(n)CC_n$$

Metriche utilizzabili come fattori di influenza:

NC - Numero totale delle classi

NRC - Numero di classi radice

NLC - Numero di classi foglia