Lez 18 (7/12)

Si entra nella fase successiva alla specifica dei requisiti, passando alla fase di **Progettazione**. Vedremo davvero la fase di progettazione nella seconda parte del corso, adesso ci focalizzeremo su una fase a carattere più gestionale che si interpone tra le due, la fase di **Pianificazione**.

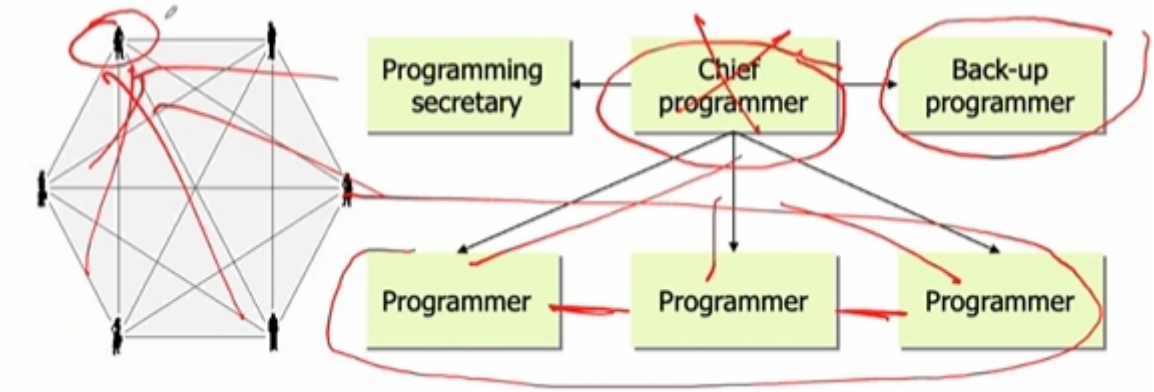
La ***fase di pianificazione*** *rientra in un processo più ampio*: la **Gestione di Progetti** Software che **comprende pianificazione, monitoraggio e controllo di persone, eventi e processi durante lo sviluppo del prodotto**.
Dal punto di vista tecnico come visto il documento più importante prodotto durante lo sviluppo di un software è il **documento di specifica dei requisiti** *in quanto guiderà sia l'attività di sviluppo che la successiva manutenzione post rilascio*.
Invece **dal punto di vista gestionale** il documento guida utilizzato durante lo sviluppo di un progetto software è **l'SPMP Software Project Management Plan**.

In generale, *la gestione efficace di un progetto software si fonda sulle "**quattro P**":*
- **Persone** == *abbiamo introdotto la disciplina di ISW come una disciplina che superasse la crisi del software, legata all'incapacità delle singole persone per quanto esperte fossero nella creazione di prodotti software di qualità*.
Nonostante ISW fornisca tecniche e strumenti per riuscire a sviluppare al meglio il progetto software, **le Persone continuano a rappresentare l'elemento più importante**. (il SEI ha addirittura elaborato oltre al Capability Maturity Model che permetteva di valutare le capacità di un'organizzazione di svilupparre software anche un *People Management Capability Maturity Model* dove le KPA sono legate alla gestione del personale (addestramento, ferie etc…))
- **Prodotto** == ***identifica le caratteristiche del software che deve essere sviluppato*** (obiettivi, dati, funzioni…)
- **Processo** == ***definisce il quadro generale di riferimento entro il quale si stabilisce il piano complessivo di sviluppo del prodotto software***
- **Progetto** == **definisce l'insieme di attività da svolgere identificando compiti, persone, tempi e costi**. *In questo senso si può pensare al progetto come una sorta di istanza del processo.*

Abbiamo già in precedenza introdotto e parlato di Prodotto e Processo, adesso ci concentreremo sul capire quanto incidono le Persone e come incidono e poi ci occuperemo della gestione del Progetto in termini di pianificazione (successiva alla fase di analisi dei requisiti) di tempi, costi e risorse.
Chiaro che si debba fare dopo la fase di specifica in quanto devo sapere esattamente COSA deve fare il software per capire COME farlo lavorando anzitutto sulla **pianificazione** (stimando il tempo, costi e risorse necessarie, tempo importantissimo soprattutto per software a contratto ma anche non (promessa rilascio al mercato e danni d'immagine se non avviene in tempo))

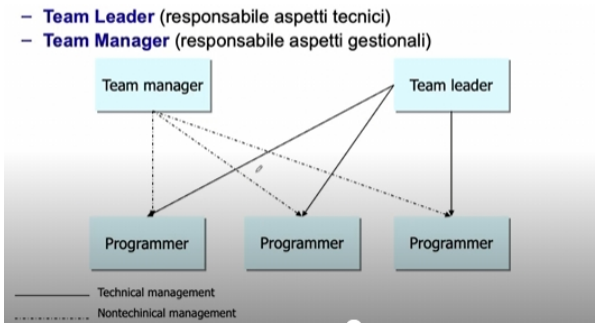
Partiamo quindi dalle **Persone**.
Problema: viene chiesto di sviluppare il prodotto software in 3 mesi e dall'analisi dei requisiti si capisce che sia necessario 1 anno/uomo (qm di effort, quantità di lavoro necessaria per svolgere una certa attività).
La soluzione immediata è farci lavorare 4 persone, ognuna 3 mesi/persona per un totale di 12 mesi/persona > 1 anno/persona (==11 mesi/persona)
La realtà empirica tuttavia ci dice che i 4 sviluppatori potrebbero impiegare un anno ottenendo un prodotto di qualità inferiore a quello risultante dal lavoro di un singolo sviluppatore.
Ciò è dovuto al fatto che si **lavora su un qualcosa di complesso, e aumentare il numero di persone significa anche aumentare il numero di interazioni, distribuire le decisioni ed avere eventuali incomprensioni**. Inoltre assegnare un compito che ha impegno pianificato 1 anno/uomo a 4 persone per farlo in tre mesi non tiene conto del fatto che ***alcuni compiti non possono essere condivisi***.
Inoltre *se si decidesse di aggiungere un ulteriore sviluppatore quando si osserva che il progetto è in ritardo allora secondo la* **Legge di Brooks** (lo stesso che aveva scritto l'articolo No Silver Bullet, che descriveva le difficoltà essenziali e accidentali nello sviluppo software) si rischierebbe di ritardare ulteriormente il progetto a causa del periodo di formazione (per mettersi in pari) e dell'aumento di interazioni.

Descriveremo *due degli approcci che permettono di organizzare team*.
- **Approccio Democratico** (orizzontale): ('70) è ***basato sul concetto di egoless programming, per cui il codice prodotto è il codice del team piuttosto che del singolo sviluppatore che ha contribuito alla singola parte di codice*** (evitando così attacchi personali quando si trova un difetto del codice).
Vantaggi *== atteggiamento positivo nella ricerca di difetti e molto produttivo in caso si affronti lo sviluppo software particolarmente difficili* (es. **ricerca**, la condivisione pari livello produce vantaggi in termini di efficacia e produttività)
Svantaggi *==* ***l'approccio non può essere imposto ma deve nascere spontaneamente, inoltre gli sviluppatori più anziani non desiderano essere valutati al pari dei più giovani*** (*difficilmente applicabile quando team di sviluppo costituito da persone con esperienza di livello differente*)
- **Approccio con Capo Programmatore** (gerarchico, verticale): questo approccio è **basato sul concetto di specializzazione** (*ogni partecipante svolge i compiti per i quali è stato formato*) e **gerarchia** (*ogni sviluppatore comunica esclusivamente con il* ***capo-programmatore*** *che dirige le attività ed è responsabile dei risultati*).
Backup Programmer *== programmatore con la stessa esperienza degli altri che fa testing e che può fungere da "backup" in caso di problemi con gli altri*
Programming Secretary *== tiene traccia di tutte le attività svolte ed è responsabile della documentazione*



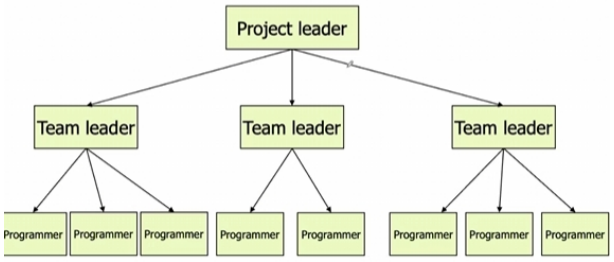
Vantaggi == **È stato dimostrato come l'approccio funzioni in domini specifici già organizzati in termini gerarchici** (*militare, difesa*), inoltre diminuisce il numero di canali di comunicazione (vedi a sx n(n-1) canali per il democratico, a destra molti meno) *e quindi diminuiscono anche i problemi in base alla legge di Brooks*
Svantaggi == **richiede personale molto esperto per ricoprire i vari incarichi**; *il capoprogrammatore è sia manager che tecnico con grande esperienza, il programmatore di backup che può dover anche sostituire il capoprogrammatore ed è responsabile di attività di testing, poi il segretario di programmazione che è responsabile della documentazione e dell'archivio di produzione.*

Questi due approcci appena visti sono agli estremi, ovviamente *ne esistono molti altri* "**in between**".
Si è assistito in particolare ad una ***evoluzione degli approcci***, per cui ad esempio nel modello verticale *si vuole distinguere il ruolo gestionale dal tecnico nel capoprogrammatore* -> si evidenziano due figure, il **Team Leader** (responsabile di aspetti tecnici) e **Team Manager** (gestionali) invece che il singolo capoprogrammatore.



Tuttavia nasce un problema: in questo approccio leader e manager non hanno canali di comunicazione. Se ad es. un programmatore vuole andare in ferie in un momento critico del progetto le chiede al team manager che gliele darà togliendo quindi una risorsa importante al team leader.
Soluzioni a ciò è identificare aree di responsabilità condivise per permettere ai due di comunicare e stratificare ulteriormente, introducendo un altro livello di gestione con un project leader responsabile di progetto e che comunica con i team leader.

- Organizzazione (scalabile) dei team per *progetti SW di dimensione medio-grande*



Ulteriore evoluzione è l'approccio che prende in considerazione anche l'approccio orizzontale, con decision making decentralizzato per cui ***si introducono canali di comunicazione tra programmatori, tra team leader sfruttando quindi i vantaggi dell'approccio orizzontale.***

È molto importante il Software Project Management nonostante spesso si pensi che per fare un buon progetto sia sufficiente la conoscenza tecnica, si è fatta un'indagine sui progetti che dal 1984 al 2016 sono stati rilasciati in tempo, in ritardo o cancellati a causa di ritardi/costi eccessivi o scarsa qualità. Tra quelli rilasciati in tempo solo il 50%, il 13% cancellati e il resto in ritardo.

Nella fase di Pianificazione l'Obiettivo è definire un quadro di riferimento generale per controllare e determinare l'avanzamento e lo sviluppo di un progetto software.
Ciò è importante in quanto l'azienda deve essere in grado di sviluppare prodotti software nei tempi e costi stabiliti, con le desiderate caratteristiche di qualità.

Le componenti fondamentali di una corretta pianificazione sono:
Scoping (*comprendere il problema e lavoro da svolgere*), **Stime** (*prevedere tempi, costi ed effort*), **Rischi** (*definire le modalità di analisi e gestione dei rischi*), **Schedule** (*allocare attività e allocare le risorse disponibili per ogni attività*), **Strategia di controllo** (avere gli strumenti per capire se si sta procedendo correttamente).
Ciò su cui ora ci focalizzeremo è la parte dedicata alle Stime.
Come fare, dopo aver prodotto il documento di specifica, a prevedere tempi, costi ed effort?

L'obiettivo delle attività di stima in generale è ridurre al minimo il grado di incertezza e limitare i rischi comportati da una stima (*quando abbiamo ad es. parlato di risk management abbiamo detto che uno dei rischi più importanti è size under estimation, cioè una stima errata della dimensione del software può portare a pianificazioni scorrette e non rispettare i tempi di consegna*).

Le **tecniche di stima** possono basarsi principalmente su:
1- **stime su progetti simili già completati** *(expert judgement by analogy*, basato semplicemente sull'esperienza e utilizzo di dati che aiutano nella stima)
2- "**tecniche di scomposizione**" (approccio bottom-up)
3- **modelli algoritmici empirici** (*più accurato*)

Le **tecniche di scomposizione** *utilizzano un approccio* ***divide et impera****, scomponendo il problema (scompongo il prodotto nelle sue componenti principali ragionando sulle stime di queste, per poi aggregare)*. Si basano su stime dimensionali (come LOC Lines of Code o FP Function Points)
I **modelli algoritmici empirici** invece *si basano su dati storici e su relazioni del tipo d=f(vi) dove d è il valore da stimare (es. costo, effort, durata) mentre vi sono le variabili indipendenti (es. LOC o FP stimati)*

Il problema di entrambe le tecniche è che richiedono come variabile indipendente (parametro da fornire in input alla tecnica) **una stima dimensionale del software** (tipicamente tramite LOC) e quindi si torna al problema della size under stimation: se sottostimo la dimensione del software fornirò un valore scorretto in input alle tecniche di pianificazione, che daranno un risultato altrettanto sottostimato. Ma noi ci troviamo subito dopo l'analisi dei requisiti, la vera dimensione del software la saprò solo dopo la fase di codifica e integrazione, *ed è difficile stimare prima LOC*. ***Sono quindi state introdotte unità di misura alternative (come i punti funzione FP) che però purtroppo non sono compatibili con diversi modelli algoritmici empirici***.

Vediamo come funziona **la tecnica di scomposizione**.
Vediamo l'esempio di un prodotto di tipo CAD. Si stima guardando componenti più piccole:

Functions	estimated LOC	LOC/pm	$/LOC	Cost	Effort (MM)
UICF	2340	315	14	32,000	7.4
2DGA	5380	220	20	107,000	24.4
3DGA	6800	220	20	136,000	30.9
DBM	3350	240	18	60,000	13.9
CGDF	4950	200	22	109,000	24.7
PCF	2140	140	28	60,000	15.2
DAM	8400	300	18	151,000	28.0
Totals	33,360			655,000	144.5

UICF gestione interfaccia utente, 2DGA e 3 analisi geometria a 2 e 3 dimensioni, DBM gestione dati (database management), CGDF strumento di computer graphics, PCF gestione periferiche, DAM per l'analisi strutturale del progetto.
Devo anzitutto fornire il numero stimato di LOC per ogni componente, difficile tipicamente fare queste stime (tipicamente si fa ciò quando si sta aggiornando un progetto es. aggiungendo funzionalità di modo che non si sottostimi troppo).

A questo punto chi fa uso della tecnica deve fornire come input anche LOC/pm (parametro di effort, produttività, **quantità di lavoro per lo sviluppo quindi righe di codice per mese/uomo**, quante righe di codice produce una persona in un mese lavorativo) e *$/LOC* (**parametro di costo, costo per riga di codice**).
Ci si accorge come in base alla componente cambi il LOC/pm, infatti ad esempio per l'interfaccia 315 e per la gestione periferiche solo 140, ciò dipende dalla difficoltà di quello che si deve fare (e nel caso del componente interfaccia inversamente proporzionale rispetto a LOC/pm, più facile costo 14 dollari per riga, nel caso delle periferiche 28 dollari per riga).
Forniti questi tre input tramite la tecnica di scomposizione **posso derivarmi le colonne di Cost e Effort (MM)** (semplicemente moltiplico $/LOC per estimated LOC per il primo e per il secondo divido estimated LOC per LOC/pm).
NB effort e costi calcolati riguardano solo la fase di progettazione (dal post specifica al rilascio).
Si ottiene la stima di un progetto con più di 33k righe di codice, che costa 655k dollari e che richiede effort di 144.5 mesi/uomo (ossia se lo sviluppasse una sola persona ci metterebbe più di 12 anni). (nb si tratta di un prodotto di dimensioni medie, e ci accorgiamo quindi che già per un progetto del genere è necessario usare un team di sviluppo).

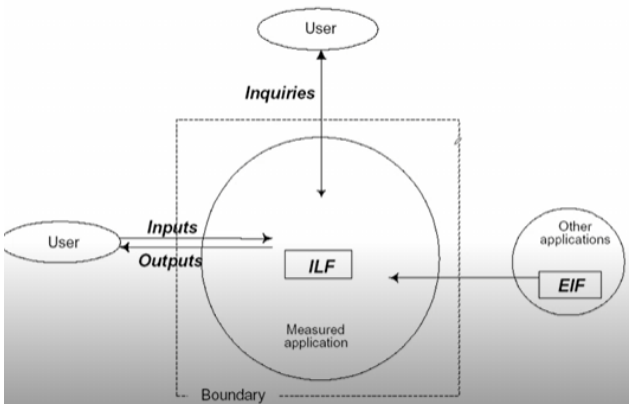
Questa tecnica funziona solo se siamo in grado di avere dati relativi a produttività LOC/pm e costo $/LOC ma soprattutto se siamo in grado di stimare accuratamente il numero di righe di codice per componente, e per farlo come anticipato è fondamentale lavorare con analogia dei progetti simili passati.
Tuttavia un altro grosso problema sta nel fatto che questi dati non servono più a nulla se arriva un cliente che ci chiede di realizzare lo stesso identico progetto già fatto in passato ma su un linguaggio diverso **(LOC infatti è intrinsecamente legato al linguaggio di programmazione utilizzato**, i linguaggi di programmazione hanno poteri espressivi diversi, vedi differenza tra lunghezza codice di 100 righe in C e il corrispondente in linguaggio macchina).
*Questa dipendenza dalle righe di codice è un problema che è stato affrontato con l'introduzione di un'unità di misura specifica svincolata dal linguaggio di programmazione ma vicina al documento di specifica: **FP**.*

Lez 19 (18/12)
Di fronte a queste difficoltà e al fatto che la maggioranza delle tecniche di stima sfrutta come um LOC (che come detto comporta il rischio di size under estimation) è *stata introdotta un'unità di misura svincolata dal linguaggio di programmazione e dalle righe di codice e molto più vicina al documento di specifica*: **Punto Funzione**.
Rappresenta un numero adimensionale (si conta il numero di punti funzione) che misura il ***numero di funzionalità*** **che il prodotto software contiene basandosi sul documento di specifica.**

Per questa ragione questa tecnica prevede la stima prima dell'implementazione (*ossia calcolo il numero di FP a partire dal documento di specifica che ho già scritto, non devo fare delle stime in funzione di ciò che potrebbe essere come per LOC*).
Un punto funzione incorpora una certa quantità standard di funzionalità che poi sarà usata come riferimento per calcolare la quantità complessiva di funzionalità.
La IFPUG (international function point user group) è l'organizzazione internazionale che fornisce il **manuale con le modalità con cui calcolare il numero di FP a partire dal documento di specifica**, il manuale è continuamente aggiornato e molto dettagliato.

Il conteggio FP è effettuato seguendo ***due step*** (a partire sempre dal doc. di specifica):
1) si calcola anzitutto **l'Unadjusted Function point Count** (**UFC**) == *si tiene conto esclusivamente della funzionalità che il prodotto deve offrire, senza guardare alla complessità tecnica*
2) si **moltiplica l'UFC con il Technical Complexity Factor TFC** ->
FP = UFCxTFC

Daremo solo qualche cenno di come funziona il meccanismo di conteggio per il calcolo di FP, sottolineando quindi il fatto di come *il manuale fornito da IFPUG sia molto dettagliato affinché il calcolo sia il più preciso possibile*.
UFC è calcolato rilevando anzitutto 5 valori (detti elementi di conteggio) dal documento di specifica:



Il cerchio grande rappresenta il software sul quale viene fatta la misura di punti funzione, tale software ha come confine Boundary (oltre il quale vi sono gli utenti e gli altri sistemi con cui il software interagisce).
Dei 5 elementi di conteggio citati prima **due fanno riferimento ai dati** (informazioni che il sistema software deve gestire o scambiare con altri) **mentre gli altri tre fanno riferimento alle interazioni con il mondo esterno** in termini di utenti del software stesso
-> **ILF** ed **EIF** sono i **due elementi di conteggio legati alle informazioni**, *ILF per informazioni interne al software* (che egli deve gestire ad es. memorizzandole in un

database) mentre *EIF sono quelle informazioni che il software scambia con altri software.*

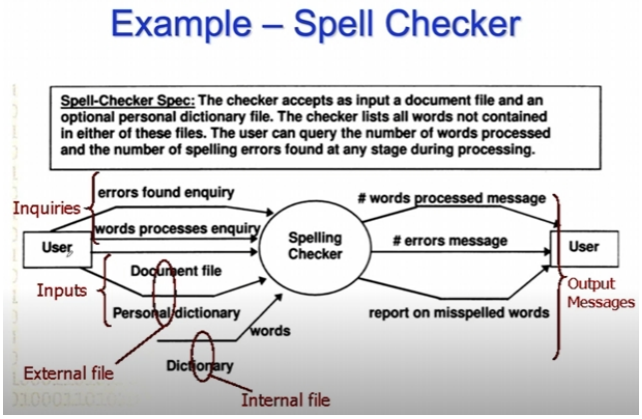
Riguardo i tre elementi di conteggio legati al mondo esterno sono **il numero di Input forniti dall'utente**, il **numero di Output** *restituiti dal software all'utente* e il **numero di Inquiries** (ossia input utente che corrispondono a uno specifico output del software).

Partiamo approfondendo gli elementi di conteggio della categoria dati.
-**Numero di ILF** (Internal Logical Files) == *gruppo di dati o informazioni di controllo generato, utilizzato o mantenuto dal sistema software* (es. dati mantenuti dal software in database)
-**Numero di EIF** (External Interface Files) == *gruppo di dati o informazioni di controllo passate direttamente o condivise tra applicazioni*
NB. *Il termine file non va inteso come tradizionale elemento nel file system ma come gruppo di elementi correlati tra loro*

Riguardo invece la categoria delle interazioni (transazioni) si devono contare:
-**Numero di EI** (External Inputs) == *elementi forniti dall'utente che descrivono dati o informazioni di controllo o output di altri sistemi che entrano nel mio software e* ***modificano lo stato degli internal logical files*** (le informazioni gestite dal software)
-**Numero di EO** (External Outputs) == *dati/informazioni di controllo prodotti dal software e forniti all'utente*
-**Numero di EQ** (External Inquiries) == *tutte le combinazioni input/output dove un input causa e genera un output immediato senza cambiare lo stato degli internal logical files* (**senza modificare le informazioni gestite dal software**)

Per capire se l'elemento fornito in input dall'utente è einquiry o einput devo quindi osservare se questo modifica o meno i dati del software (es. inserisco un nuovo cliente nel videostore per creare la sua carta -> einput oppure il supporter in telemarketing mi dice che ha cambiato tel -> einput se invece uso il software per cercare la mail di uno studente -> einquiry)

Vediamo un semplice esempio di applicazione



Si fa riferimento di un software il cui documento di specifica sia racchiuso in quel rettangolo sopra. Si tratta di un software che controlla gli errori ortografici nei testi. Il software accetta come input un documento e un dizionario personale opzionale e sceglie come parole sbagliate tutte quelle che non fanno parte del suo dizionario interno e del dizionario personale opzionale fornito.
Si immagina che vengano inviati documenti molto grandi, quindi durante la fase di processing l'utente può chiedere al sistema quante parole sono state processate fino a quel momento e quanti errori sono stati trovati.
A questo punto per contare il numero di elementi di conteggio riformulo il grafico di prima, il software confinato nello Spelling Checker e poi gli utenti esterni.
Contiamo anzitutto il numero di file Interni (dati che lo spell checker contiene internamente e utilizza in fase di esecuzione) -> dizionario interno (1).
I file esterni sono invece il documento su cui fare spell checking e un eventuale dizionario (2).
Le inquiries sono a tempo di esecuzione il numero di parole processate e il numero di errori trovati (2). Gli output prodotti sono il report finale con tutte le parole non parte dei dizionari e gli eventuali output delle 2 inquiries (3).
Gli input forniti dall'utente sono di nuovo il document file e il personal dict (2).
Importante puntualizzare che non possono esserci elementi ripetuti in categorie diverse! In questo caso infatti gli input forniti dall'utente non rappresentano il document e il personal dict in sé, ma il nome che li rappresenta (es. percorso file) mentre parlando di file esterni si intende il file nella sua interezza necessario per far funzionare il software.

Una volta ottenuti questi numeri EI=2, EO=3, EQ=2, EIF=2, ILF=1 sfrutto una tabella fornita dal manuale di conteggio dove inserire i numeri contati nella prima colonna. Per ognuno di essi decido la corrispettiva complessità funzionale, e in base ad essa decido per quanto moltiplicarli.

Example - *UFC*

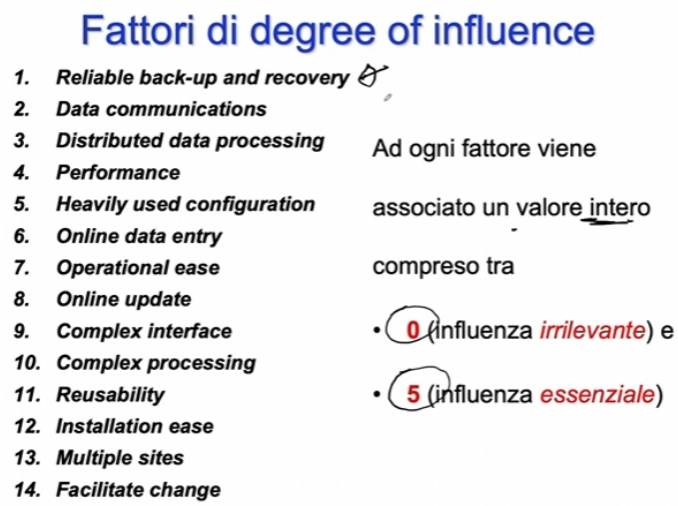
- Assume average complexity weights for each element
- UFC = 55

Function Type	Functional Complexity		Complexity Totals	Function Type Totals
ILFs	___ Low	X 7 =	___	
	1 Average	X 10 =	10	
	___ High	X 15 =	___	10
EIFs	___ Low	X 5 =	___	
	2 Average	X 7 =	14	
	___ High	X 10 =	___	14
EIs	___ Low	X 3 =	___	
	2 Average	X 4 =	8	
	___ High	X 6 =	___	8
EOs	___ Low	X 4 =	___	
	3 Average	X 5 =	15	
	___ High	X 7 =	___	15
EOs	___ Low	X 3 =	___	
	2 Average	X 4 =	8	
	___ High	X 6 =	___	8
		Total Unadjusted Function Point Count		55

NB. la complessità funzionale è decisa sempre con l'aiuto del manuale.
es. immaginiamo che le ILF siano le tabelle di un database. Tre tabelle, una con 20 colonne (ogni record della tabella ha 20 dati), una con 5 colonne e una di 10 colonne. Il manuale di conteggio potrà fornirmi dettagli riguardo ILF in questo caso, ad esempio potrebbe dirmi se una tabella ha numero di colonne inferiore a 10 allora Low, altrimenti etc…
Quindi in realtà i numeri degli elementi di conteggio andrebbero divisi ulteriormente (es. EO=3 potrebbe avere 2Low e 1Average).

Abbiamo quindi finalmente calcolato UFC (unadjusted perché finora abbiamo considerato questi elementi solo dal punto di vista della loro complessità funzionale!)
Difficilmente ci si sbaglia di troppo perché il manuale di conteggio è molto dettagliato.

Per ottenere gli FP definitivi devo però considerare *anche la difficoltà tecnica oltre a quella funzionale*. La difficoltà tecnica è calcolata a partire da 14 fattori, detti Fattori di Degree of Influence.
Il nome deriva dal fatto che ognuno di questi fattori può avere influenza più o meno forte nel caso specifico del software dove sto effettuando il calcolo degli FP.
Ad ognuno dei fattori è associato un valore intero compreso tra 0 e 5, dove 0 == influenza irrilevante nel caso specifico del mio software e 5 == influenza essenziale.



I 14 fattori sono tutti quei fattori che potrebbero influenzare l'applicazione dal punto di vista tecnico es. *se l'applicazione ha necessità di effettuare backup e recovery in modo affidabile* (nel caso spell checker è 0, nel caso di un qualsiasi sistema d'archiviazione forse anche 5), *data communications*, *distributed data processing* (è software locale o distribuito, nel caso spell checker 0 in quanto lo eseguo nel mio pc), *prestazioni*, **online data entry** (è significativo l'ingresso dei dati o meno), facilità d'uso, …, *riusabilità,* facilità d'installazione, se è necessario installarlo su più siti, *facilità di modifica* (manutenibilità).

Una volta associati i valori interi per ognuno dei 14 fattori, posso calcolare **TCF** come segue:

- Each component is rated from 0 to 5, where:
  - 0 Not present, or no influence
  - 1 Incidental influence
  - 2 Moderate influence
  - 3 Average influence
  - 4 Significant influence
  - 5 Strong influence throughout
- The TCF can then be calculated as:

$$TCF = 0.65 + 0.01 \sum_{J=1}^{14} F_j$$

- The TCF varies from 0.65 (if all F_j are set to 0) to 1.35 (if all F_j are set to 5) ➔ ± *35% adjustment*

+-35% adjusted perché se metto a tutti gli Fj 5 allora ottengo 0.65 + 0.7 = 1.35 mentre se metto 0 ottengo 0.65.
Si parla di aggiustare perché UCF verrà "aggiustato" di questa percentuale in avanti o indietro per via della moltiplicazione, non sarà mai stravolto ma sistemato in funzione della difficoltà tecnica.

Example – *TCF and FP*

- Suppose that:

F1 = 3	Reliable back-up and recovery	F7 = 3	Operational ease
F2 = 3	Data communications	F8 = 3	Online update
F3 = 0	Distributed data processing	F9 = 0	Complex interface
F4 = 5	Performance	F10 = 5	Complex processing
F5 = 0	Heavily used configuration	F11 = 0	Reusability
F6 = 3	Online data entry	F12 = 0	Installation ease
		F13 = 0	Multiple sites
		F14 = 3	Facilitate change

- then

$$TCF = 0.65 + 0.01(18+10) = 0.93$$

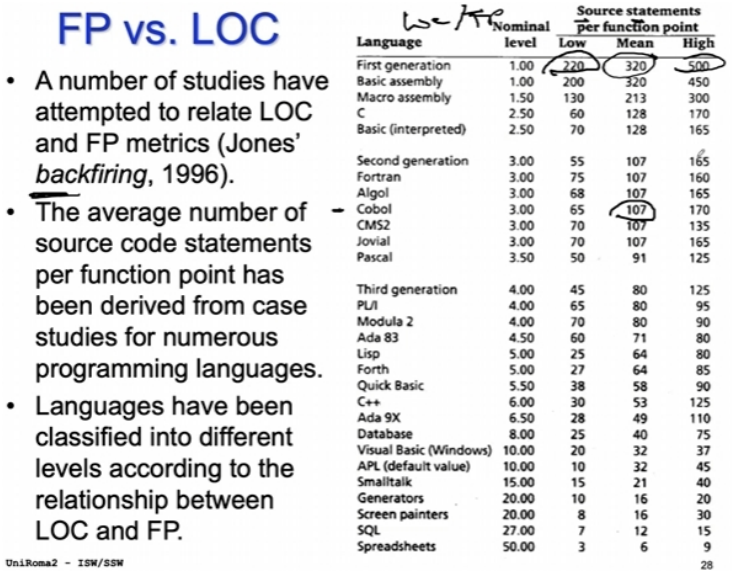
- and

$$FP = 55 \times 0.93 \approx 51$$

FP ampliamente utilizzato attualmente anche nei bandi di gara, nel contratto si forniscono gli FP di modo che chi risponde al bando di gara possa quantificare la dimensione tecnico-funzionale del software da realizzarsi e le corrispettive offerte.

Tuttavia resta un problema, le tecniche di stima di tempi costi ed effort maggiormente utilizzate non si basano su FP ma su LOC.
Come stimare LOC?
FP calcola il numero di punti funzione facendo riferimento al singolo punto funzione come misura di una quantità di funzionalità. **La stessa funzione realizzata con un certo linguaggio di programmazione avrà uno specifico numero di LOC diverso dagli altri.** Per tradurre quindi FP in LOC devo scegliere un determinato linguaggio.

Sono state quindi sviluppate delle tabelline dette di backfiring. Indiana Jones ha preso molti progetti con relativi documenti di specifica, per ogni documento ha calcolato il numero di FP e le corrispettive LOC, dove ogni LOC fa riferimento a uno specifico linguaggio di programmazione in base al progetto.
Vediamo solo un estratto visto che Jones è furbo e le vende.
Per ogni linguaggio è fornito il source statement per punti funzione (a quante LOC in uno specifico linguaggio di programmazione corrisponde il singolo punto funzione).
Chiaramente analisi statistica quindi lower bound, numero medio e upper bound.
(*) **Quindi avendo a disposizione la tabella ciò che si fa è calcolare FP del mio doc di specifica, calcolare LOC corrispondenti tramite la tabella e fornirle in input alla mia tecnica di stima di effort e tempi che sarà accurato visto che ho preso in considerazione FP e corrispondente LOC via backfiring.**



Language	Nominal level	Source statements per function point: Low	Mean	High
First generation	1.00	220	320	500
Basic assembly	1.00	200	320	450
Macro assembly	1.50	130	213	300
C	2.50	60	128	170
Basic (interpreted)	2.50	70	128	165
Second generation	3.00	55	107	165
Fortran	3.00	75	107	160
Algol	3.00	68	107	165
Cobol	3.00	65	107	170
CMS2	3.00	70	107	135
Jovial	3.00	70	107	165
Pascal	3.50	50	91	125
Third generation	4.00	45	80	125
PL/I	4.00	65	80	95
Modula 2	4.00	70	80	90
Ada 83	4.50	60	71	80
Lisp	5.00	25	64	80
Forth	5.00	27	64	85
Quick Basic	5.50	38	58	90
C++	6.00	30	53	125
Ada 9X	6.50	28	49	110
Database	8.00	25	40	75
Visual Basic (Windows)	10.00	20	32	37
APL (default value)	10.00	10	32	45
Smalltalk	15.00	15	21	40
Generators	20.00	10	16	20
Screen painters	20.00	8	16	30
SQL	27.00	7	12	15
Spreadsheets	50.00	3	6	9

Inoltre prima di queste tabelline si distinguevano i linguaggi per generazione in base al loro potere espressivo, più ad alto livello più espressivi.
Con queste tabelle grazie al calcolo del numero di LOC possiamo associare valori quantitativi per rappresentare la capacità di espressione, da qui il Nominal Level della tabella che aumenta all'aumentare delle generazioni.
C++ ha ad es. lvl nominale 6 -> se un'applicazione scritta in c++ ha 1000 righe di codice, tradotta in linguaggio macchina (che ha lvl nominale 1) avrà 6000 righe di codice. Per Cobol invece ha il doppio in quanto lvl nom 3 -> 2000 LOC

Quindi tornando alla domanda iniziale, **come stimare tempo effort e costo a partire dal documento di specifica**, devo *o ragionare per analogia* (***ma devo avere dati affidabili e usare sempre lo stesso linguaggio***) oppure più accuratamente quanto detto in (*). ***Si noti che con il secondo metodo posso anche essere più preciso nel caso in cui alcuni punti funzione siano implementati in linguaggi diversi, calcolo LOC separati per entrambi i linguaggi secondo la tabella e poi li sommo***.

Lez 20 (21/12)

Calcolati i FP dal doc di specifica e i corrispettivi LOC possiamo utlizzare oltre che la tecnica di tabelle di scomposizione anche i modelli algoritmici.
Vediamo di questi ultimi in particolare **COCOMO** (**COnstructive COst MOdel**), *modello introdotto da Barry Boehm nel 1981 (lo stesso del modello a spirale).*

Questo modello serve a determinare l'effort a partire da LOC e a partire dall'effort calcolare anche la durata e i costi di sviluppo.
Modello conveniente quindi in quanto lega effort (variabile dipendente) a LOC (variabile indipendente) fornendo quindi una formula che mi permette di ricavare l'effort.

Per utilizzare COCOMO si compiono ***due scelte***: *una relativa al **modello da usare*** e *l'altra legata al **modo di sviluppo del prodotto***.
Nel caso del **modello** ne esistono *tre che cambiano in base al livello di conoscenza del nostro software*: **Basic** **per stime iniziali** (siamo alla fine della fase di specifica e non abbiamo fatto ragionamenti sulla parte progettuale), **Intermidiate** da usare dopo aver suddiviso il sistema in sottinsiemi (si ha un'idea dal punto di vista architetturale) e **Advanced** usato dopo aver operato una decomposizione modulare (si sono identificati quindi i moduli facenti parte ciascun sottosistema e le relazioni tra essi..).
Il **modo di sviluppo** invece *misura il livello intrinseco di difficoltà di sviluppo e fa riferimento alla dimensione del software*: **Organic** (prodotti di piccole dimensioni), **Semidetached** (intermedie), **embedded** (prodotti complessi e di grandi dimensioni).

Una volta scelto modello e modo, dovrò dare in input la dimensione del software a COCOMO in termini di KLOC (chiloLOC == 10^3 LOC).
Analizzeremo la versione originale, basti sapere che nel '95 è stato introdotto COCOMO II, più flessibile e sofisticato.
Prendiamo per vedere il funzionamento il caso di modello Intermidiate e modo Organic.
NB Boehm ha sviluppato il modello a partire da progetti sviluppati nella NASA, egli ha pubblicato un libro con un capitolo con una procedura di calibrazione che permette ad ogni organizzazione di trovare la funzione che meglio si adatta alle caratteristiche dei progetti software portati avanti dall'organizzazione.

In generale il calcolo dell'effort è effettuato con formule del tipo
Effort Nominale = a(KLOC)^b, ciò che cambia è sostanzialmente il valore di a e b in funzione del modello e modo.
Nel caso Intermediate Organic allora a = 3.2 e b = 1.05.
Il numero restituito in output non è puro, ma misurato in MM (mesi/persona).
Si parla di Effort Nominale in quanto questo valore non tiene conto di caratteristiche sul costo (simile a quanto visto per i punti funzione con i fattori di degree influence)

Esempio d'uso di COCOMO

Modello *Intermediate*, modo *organic*

- ***Passo 1***

  Determinare l'**effort nominale** usando la formula:

  $$\textit{effort nominale} = 3.2 \times (KLOC)^{1.05}\ MM$$

  Esempio:

  $$3.2 \times (33)^{1.05} = 126\ MM$$

- ***Passo 2***

  Ottenere la stima dell'effort applicando un fattore moltiplicativo *C* basato su 15 ***cost drivers***:

  $$\textit{effort} = \textit{effort nominale} \times C$$

  Esempio:

  $$126 \times 1.15 = 145\ MM$$

- *C (cost driver multiplier)* si ottiene come *produttoria* dei *cost driver* c_i. Ogni c_i determina la complessità del fattore *i* che influenza il progetto e può assumere uno tra più valori assegnati con variazioni intorno al valore unitario (valore nominale)

Si osserva che passando 33 KLOC (33.000 righe di codice, come per l'esempio di stima con la tecnica di scomposizione) mentre nel caso di scomposizione ottenevamo 144.5 MM con COCOMO 126MM però di effort nominale.
L'effort nominale deve essere "aggiustato" moltiplicandolo per un fattore moltiplicativo C il cui valore (similmente a quanto visto per il calcolo di FP) è basato su 15 cost drivers.
In particolare C si ottiene come produttoria dei cost driver Ci. Ogni Ci determina la complessità del fattore i che influenza il progetto e ognuno di essi può assumere un valore che si scosta in più o in meno dal valore unitario (nominale, se scelgo tutti i Ci con valore unitario allora effort = effort nominale).

Approfondiamo questi cost driver e vediamo come si calcolano.
Nel caso di FP erano fattori che avevano influenza sulla complessità tecnica del prodotto e che "aggiustavano" di +-35% UnadjustedFC.

Tabella di cost driver (Intermediate COCOMO)

Cost Drivers	Rating					
	Very Low	Low	Nominal	High	Very High	Extra High
Product Attributes						
Required software reliability	0.75	0.88	1.00	1.15	1.40	
Database size		0.94	1.00	1.08	1.16	
Product complexity	0.70	0.85	1.00	1.15	1.30	1.65
Computer Attributes						
Execution time constraint			1.00	1.11	1.30	1.66
Main storage constraint			1.00	1.06	1.21	1.56
Virtual machine volatility*		0.87	1.00	1.15	1.30	
Computer turnaround time		0.87	1.00	1.07	1.15	
Personnel Attributes						
Analyst capabilities	1.46	1.19	1.00	0.86	0.71	
Applications experience	1.29	1.13	1.00	0.91	0.82	
Programmer capability	1.42	1.17	1.00	0.86	0.70	
Virtual machine experience*	1.21	1.10	1.00	0.90		
Programming language experience	1.14	1.07	1.00	0.95		
Project Attributes						
Use of modern programming practices	1.24	1.10	1.00	0.91	0.82	
Use of software tools	1.24	1.10	1.00	0.91	0.83	
Required development schedule	1.23	1.08	1.00	1.04	1.10	

*For a given software product, the underlying virtual machine is the complex of hardware and software (operating system, database management system) it calls on to accomplish its task.

In questo caso invece i cost driver hanno un rating: valore nominale 1.00 (infatti poi li metto in produttoria e se sono tutti 1 allora effort = effort nominale), il rating scende o sale a seconda dei casi, *se scende ho risparmi in termini di costo se invece sale avrò costo aggiuntivo.*
I 15 fattori sono divisi in 4 gruppi (legati a caratteristiche di **Prodotto**, **Piattaforma d'esecuzione**, **Personale** e **Progetto**).
Required Software Reliability == l'applicazione ha requisiti di affidabilità stringenti o no? Se non ne ha posso scegliere Very Low -> risparmio sul costo perché il prodotto non ha quella esigenza, se invece alti posso aumentare molto l'effort!
Poi *dimensione database*, *complessità di prodotto* (per cui è definita anche Extra High con influenza Ci 1.65), *vincoli di tempo d'esecuzione* (es. per software real time), *memorizzazione*, *volatilità virtual machine*, *tempo di risposta turnaround*.
Poi per il calcolo dell'effort contano anche le *caratteristiche del personale* -> *competenze analisti software* (**in questo caso se le competenze sono alte allora si risparmia sui costi! 0.71 Very High**), *conoscenza del dominio applicativo*, *competenza programmatori*, *conoscenza del sistema operativo* dove si definisce il software, *conoscenza linguaggio programmazione* e lo stesso vale per gli attributi di progetto: se uso di pratiche moderne di programmazione allora l'effort diminuisce e lo stesso per l'uso di strumenti software.
L'ultimo cost driver, "*Required development schedule*" **è l'unico che fornisce un valore maggiore di 1 sia a sx che a dx** -> *conviene stare al valore nominale*.
***Questo attributo indica il fatto che il mio progetto abbia o meno tempo stringente di consegna, sia che abbia tempi molto stringenti che non l'effort aumenta. Il motivo intuitivamente sta nel fatto che se devo fare qualcosa entro poco tempo mi ci metto tanto quindi grande effort, se invece ho tanto tempo mi ci metterò comunque parecchio per cercare la soluzione migliore*.**

Applicando il prodotto per il fattore C non si modificherà mai l'effort di un ordine di grandezza, ma similmente a quanto visto per l'aggiustamento di UFC si "aggiusterà" l'effort nominale di un tot.

Per capire quale rating conviene scegliere ci sono parametri standard da seguire, es:

	Ratings					
Cost Driver	Very Low	Low	Nominal	High	Very High	Extra High
Product attributes						
RELY	Effect: slight inconvenience	Low, easily recoverable losses	Moderate, recoverable losses	High financial loss	Risk to human life	
DATA		$\frac{\text{DB bytes}}{\text{Prog. DSI}} < 10$	$10 \leq \frac{D}{P} < 100$	$100 \leq \frac{D}{P} < 1000$	$\frac{D}{P} \geq 1000$	
CPLX	*See next slide*					
Computer attributes						
TIME			≤ 50% use of available execution time	70%	85%	95%
STOR			≤ 50% use of available storage	70%	85%	95%
VIRT		Major change every 12 months Minor: 1 month	Major: 6 months Minor: 2 weeks	Major: 2 months Minor: 1 week	Major: 2 weeks Minor: 2 days	
TURN		Interactive	Average turnaround <4 hours	4–12 hours	>12 hours	

Vediamo come per es. se dalla reliability dipende la vita delle persone allora very high. Riguardo la dimensione si calcola rapporto dimensione database e numero d'istruzioni, se minore di 10 low etc…

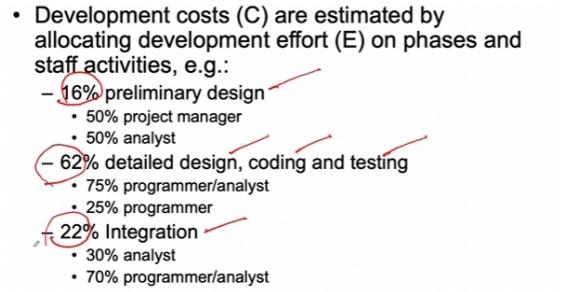
La cosa utile di COCOMO è che oltre che stimare l'effort mi permette di determinare anche la durata del progetto.
Sapendo che un software ha un effort di certi MM e sarà sviluppato da un certo team di persone, COCOMO permette di stimare il tempo che impiegheranno a svilupparlo usando ancora un'altra formula del tipo ***Tempo = c(E)^d*** *dove* ***c*** *e* ***d cambiano a seconda del modello e del modo.***

- Stima del tempo T alla consegna (product delivery):
  - Modo *organic* $T = 2.5\ E^{0.38}$ (months M)
  - Modo *semi-detached* $T = 2.5\ E^{0.35}$
  - Modo *embedded* $T = 2.5\ E^{0.32}$

Anche qui le unità di misura hanno significato preciso: l'effort dato in input deve essere MM e il valore di durata restituito è espresso in mesi (*tempo alla consegna, dal momento in cui inizio a usare COCOMO quindi tipicamente dopo analisi requisiti fino alla data di rilascio del prodotto al cliente o al mercato*, **non è quindi il tempo di vita del software ma di sviluppo a partire dal momento in cui uso COCOMO**).

La slide suggerisce come l'esponente per organic è 0.38, semi-detached 0.35 e embedded 0.32. Non possiamo però concludere subito guardando questa formula che impiego meno tempo a sviluppare un embedded rispetto a un organic, in quanto l'effort che gli pongo in input è totalmente diverso (per embedded molto superiore).

Ciò che ci manca ancora da stimare è il costo. *Questo valore si ottiene facilmente partizionando l'effort: infatti le persone che sviluppano il progetto ovviamente svolgono ruoli diversi.*
Si divide anzitutto lo sviluppo (post analisi requisiti) in tre parti: ***progettazione preliminare***, ***progettazione di dettaglio*** (codifica e testing), infine ***integrazione***.

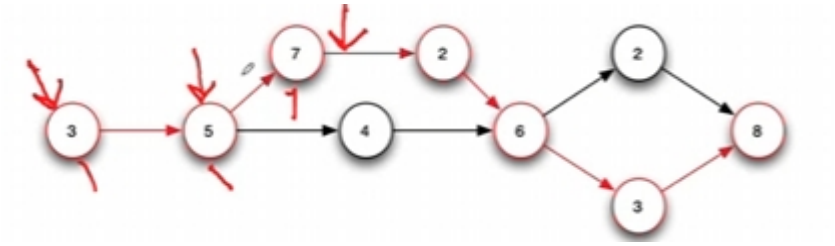


Dell'effort restituito da COCOMO ad es. 16% lo dedico al primo, 62% al secondo e 22% all'ultimo. Dopodiché ragiono sul personale, partendo dalla progettazione preliminare e così via per le altre fasi.
Per ottenere il costo finale farò i calcoli sapendo il costo MM per ogni membro del personale (non è lo stipendio ma il costo totale per l'organizzazione, chiaramente comprende lo stipendio): es. se COCOMO mi darà 100 MM tot, allora 16MM per progettazione preliminare di cui 8 MM per il PM e gli altri 8 per l'analista.
Sapendo che il PM ad es. mi costa 8k MM allora 64k solo per metà della progettazione preliminare e così via fino ad arrivare al costo complessivo.

Quindi in generale parto dal doc di spec, calcolo FP, passo a LOC, trasformo in KLOC, ottengo l'effort nominale, poi eventuale calcolo di effort reale e poi calcolo durata e costi.
Conviene imparare a memoria la formula E = 3.2 x (KLOC)^1.05 per l'effort nominale e T = 2.5 E^0.38 per stimare il tempo per modello intermediate e modo organic, usandole in generale non ci si sbaglierà più di tanto nella stima (ricordale per l'orale).

Nella parte di Pianificazione un'altra attività molto importante è quella relativa alla **Pianificazione Temporale** *per poter monitorare o controllare l'esecuzione del progetto.*
La pianificazione temporale consiste, dopo aver scelto il modello di processo, i task da eseguire, stimato costi, effort e durata, **nel definire una rete di quei task** (che NON sono tutti indipendenti!) *in base ai seguenti principi*:
- **Ripartizione** (scomposizione del processo e prodotto in parti (task e funzioni) di dimensioni ragionevoli
- **Interdipendenza** (identificazione delle dipendenze reciproche di task)
- **Allocazione delle risorse** (identificare numero di persone, effort, data inizio/fine da assegnare a ogni task)
- **Responsabilità definite** (individuare le responsabilità del personale associate a ciascun task)
- **Risultati previsti**
- **Punti di Controllo** (milestone, identificare momenti in cui è possibile capire se il progetto procede come pianificato)

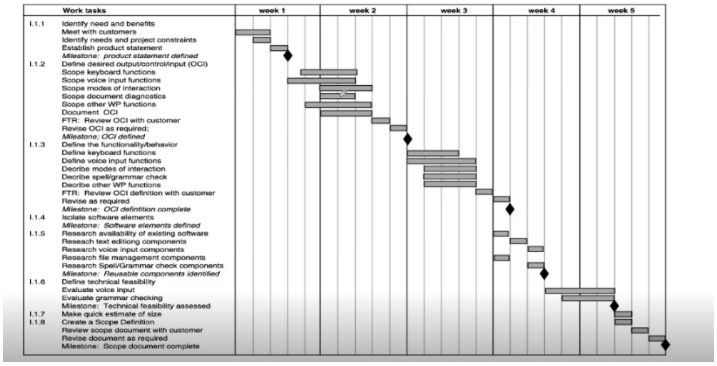
Gli strumenti che si utilizzano per operare sulla pianificazione temporale sono due:
- **Diagrammi PERT** (Program Evaluation and Review Technique) grazie al quale definiamo la rete di task come un grafo dove nodo = task e arco = legame di precedenza. Vediamo in figura il primo task che ha durata 3, una volta terminato posso passare al secondo che ha durata 5 etc…



Dal diagramma posso determinare il cammino critico (sequenza di task che determina la durata minima del progetto, il cammino più lungo), stimare il tempo di completamento per ciascun task (es. posso stimare che il task di durata 7 sopra perché venga completato in totale ci metterò 3+8+7=18) e i limiti temporali di inizio e fine di ciascun task.
In questo caso il cammino critico è segnato dal cammino in rosso, accumulare un ritardo su uno di questi task significa portarsi dietro un ritardo per l'intero progetto.

Ciò che PERT non fornisce direttamente informazioni precise a livello temporale (non so quando un progetto inizia e finisce a livello calendariale), in questo senso si sfrutta la Carta di GANTT che è una pianificazione calendariale delle attività da svolgere.
Poiché in GANTT invece non appaiono le relazioni di precedenza tra task, viene integrata con PERT per fornire informazioni complete.

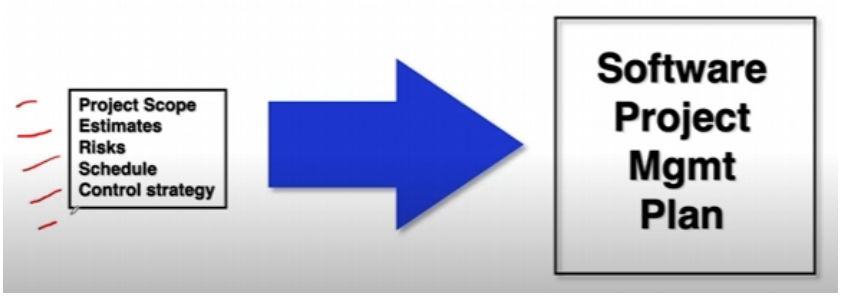
Carta di Gantt
- diagramma a barre che consente di visualizzare l'allocazione temporale dei task
- non appare nessuna indicazione dei legami di precedenza, quindi viene integrata con un diagramma PERT



Sulle righe le attività da svolgere, ogni colonna è un blocco da 5 blocchi più piccoli che corrispondono ai giorni lavorativi (un blocco da 5 è quindi una settimana lavorativa), rombetto == milestone.

Tutti questi strumenti (COCOMO, tabelle di scomposizione, GANTT, PERT così come tutta la parte di risk management) fanno parte di un processo più ampio che abbiamo detto essere il **Software Project Management**, *che è guidato da un documento fondamentale*: **SPMP** (software project management plan).

In SPMP vanno tutti questi strumenti che abbiamo elencato affinché si abbia una buona attività di gestione del progetto software.



Come nel caso del documento di specifica per l'analisi dei requisiti, questo documento non si crea da zero ma si utilizzano appositi template.
Importante sapere che ne esistono di diversi tipi, ma come per il documento di specifica questi template ovviamente includono le stesse informazioni solo organizzate in modo diverso (per approfondire vedi immagini slide pg 45).

FINE. (primo modulo)
(per esercizio assegnato vedi lezione 8 gennaio)