

L'esigenza di introdurre un approccio ingegneristico nel mondo software nacque con il passaggio dallo sviluppo di programmi (codificati) allo sviluppo di prodotti (necessità di essere progettati, analizzati e soltanto al termine codificati).

L'introduzione di tale approccio (disciplinato e rigoroso) ha portato a notevoli vantaggi ma non ha completamente risolto la crisi del software.

Difficoltà peculiari del software divise in due gruppi separati:

- **Accidentali** (e quindi **superabili con il progresso tecnologico**)

- **Essenziali** (facenti parte della natura stessa del software)

Concetto di ciclo di vita del software:

intervallo temporale in cui un prodotto software vive: dal momento in cui questo viene **creato** fino al momento in cui viene **dismesso**.

In questo periodo il software attraversa tre stadi/macrofasi:

sviluppo, **uso/manutenzione** e **dismissione**.

Nella fase di sviluppo il software passa dall'essere idea astratta a prodotto.

Questo stadio è a sua volta diviso in ulteriori fasi:

definizione e analisi requisiti (**specifica**), **progettazione**, **codifica** e **rilascio** per l'utilizzo.

Durante l'utilizzo il software subirà un numero significativo di modifiche.

Bisogna essere in grado di individuare i difetti più grandi del prodotto, portarli avanti infatti può comportare un costo di fino a due ordini di grandezza superiore rispetto al costo speso se li avessimo identificati immediatamente.

Da qui quindi l'importanza del testing, che si articola in:

- **Verifica**: controllo di correttezza

- **Validazione**: controllo del fatto che ciò che è stato realizzato corrisponda a quanto era atteso.

Poiché l'obiettivo dell'ingegneria del software è realizzare **prodotti di qualità**, ossia prodotti che svolgono correttamente la propria funzione, le attività di verifica e di validazione sono molto importanti.

Tuttavia, per questioni di comodità (requisiti difficili da validare durante lo sviluppo), **la validazione avviene spesso soltanto alla fine del ciclo di sviluppo**.

Nessuna attività di testing potrà mai tuttavia garantire che il prodotto sia totalmente privo di difetti. L'obiettivo non è infatti questo, ma **riuscire a scovarne il maggior numero possibile**.

In questo senso la **metrica DRE** (**Defect Removal Efficiency**) fa riferimento alla *percentuale di difetti trovati prima del rilascio del prodotto software*.

Come si misura?

Se ad es. il team di sviluppo trova 900 difetti prima del rilascio e gli utenti ne rilevano 100 in un intervallo di tempo standard dopo il rilascio (tipicamente 90 giorni), allora il valore di DRE è pari al 90%.

Ulteriori difficoltà non superabili con il progresso tecnologico e strumenti più avanzati in quanto fanno parte della natura stessa del software (dette **Essenziali**, “no silver bullet” by Brooks).

Quattro difficoltà:

- **Complessità** (numero possibile di configurazioni, stati delle variabili cresce esponenzialmente)
- **Conformità** (tipicamente il software deve sempre conformarsi e adattarsi ad ambienti preesistenti hardware e non il contrario)
- **Cambiabilità** (il software tipicamente subisce modifiche sostanziali durante il periodo d'uso)
- **Invisibilità** (il software in esecuzione è di per sé invisibile, non ha proprietà geometriche. Ciò comporta importanti conseguenze dal punto di vista dell'affidabilità, non possiamo vederlo nella sua interezza come un ponte).

Le difficoltà essenziali hanno un effetto sugli aspetti di costo del sw:

- **costo verso dimensione** (size, LOC lines of coding. Il costo non cresce linearmente con le righe di codice come vedremo)
- **costo verso repliche** (costo di costruzione di una copia del prodotto sw appena sviluppato, è pari a zero!)
- **costo verso ampiezza di mercato** (mercato più ampio implica più utenti da dover soddisfare -> costi maggiori)

Esiste una relazione tra costo (inteso come costo nell'intero ciclo di vita) e dimensione del software: **il costo è proporzionale al quadrato della dimensione** ($C=aS^2$) -> fare due prodotti di size $S/2$ costa meno che farne uno di size S !

Ma realizzare due prodotti di size $S/2$ e poi unirli in uno di size S non costa esattamente la metà di farne uno direttamente di size S (metterli insieme ha un costo).

Creare una o più copie di un sw ha costo 0.

Ciò che tipicamente compriamo quando acquistiamo un software non è chiaramente il software nella sua interezza (progettazione, codice esplicito) ma

solo l'eseguibile e si paga di fatto la licenza di utilizzo. Freeware se gratis. Quando viene rilasciato l'intero prodotto si parla di open-source.

Se voglio vendere un prodotto di size doppia rispetto ad uno venduto ad un prezzo p, allora:

- o richiedo un prezzo 4 volte superiore a parità di ampiezza di mercato (crescita esponenziale di costo!)
- o necessito di un'ampiezza di mercato 4 volte maggiore per venderlo a pari prezzo rispetto al vecchio prodotto (vendo a 4 volte gli utenti di prima)

Definizioni:

- **Prodotto Sw** == Codice + Documentazione
- **Artefatto** == Prodotto Sw intermedio (non eseguibile, come il documento requisiti, di specifica o di progetto)
- **Codice** == prodotto Sw finale (artefatto considerato eseguibile)
- **Sistema Sw** == insieme organizzato di prodotti Sw
(ulteriore significato: configurazione hardware/software per l'esecuzione di un prodotto)
- **Cliente** == colui che ordina/commissiona il prodotto Sw
- **Sviluppatore** == soggetto che produce il Sw
- **Utente** == soggetto che usa il Sw
- **Sw a contratto** == cliente e sviluppatore sono soggetti differenti
(software commissionato da qualcuno, NB in generale il software per il mercato non è quello a contratto in quanto non è commissionato da qualcuno!)
- **Sw interno** == software in cui cliente e sviluppatore fanno parte della stessa organizzazione (es. Nasa)

Aspetti di Affidabilità (Sw Reliability)

L'affidabilità, informalmente, rappresenta la fiducia che riponiamo in un prodotto software.

Formalmente invece rappresenta la probabilità che il prodotto software lavori correttamente in un determinato intervallo temporale (detto **Mission Time**).

Affinché il software lavori correttamente non deve manifestare malfunzionamenti o guasti. In particolare si distinguono i termini di:

- **Errore** == azione errata di chi (per ignoranza, distrazione etc..) introduce un difetto nel prodotto sw
- **Difetto** (defect) == anomalia presente nel prodotto sw
- **Guasto** (failure) == comportamento anomalo del prodotto sw dovuto alla presenza di un difetto

I guasti avvengono a causa di difetti (bug) a loro volta finiti nel prodotto software per errore.

Chiaramente non è detto che, se ho un difetto nel prodotto sw, allora si avrà necessariamente un guasto durante l'utilizzazione.

Un prodotto software con molti difetti è poco affidabile, ed è chiaro che l'affidabilità del prodotto migliora riducendo il numero di difetti.

Tuttavia non è semplice come sembra, esiste infatti una **relazione non-semplificata tra affidabilità osservata (dall'utente) e numero di difetti latenti**. Eliminare infatti difetti da parti del prodotto raramente utilizzate ha piccoli effetti sull'affidabilità osservata.

Vale in particolare la **regola 10-90**:

esperimenti condotti su programmi di notevoli dimensioni (milioni di righe di codice) mostrano che **il 90% del tempo di esecuzione totale è speso eseguendo solo il 10% delle istruzioni**.

Questo 10% è detto **core** (nucleo) del programma (se si rimuove quindi un difetto non nel core l'affidabilità osservata resterà pressoché la stessa).

Quindi il miglioramento dell'affidabilità per l'eliminazione di un difetto dipende dalla localizzazione di quest'ultimo (se nel nucleo o meno).

L'affidabilità osservata dipende quindi *da come viene utilizzato un prodotto* (in termini tecnici, dal suo profilo operativo **Operational Profile**).

Operational Profile descrive proprio quali funzioni sono utilizzate e quant spesso

Poiché utenti diversi usano il software secondo profili operativi diversi allora i difetti che si manifestano per uno potrebbero mai manifestarsi per un altro. Quindi, in definitiva, **l'affidabilità di un prodotto Sw dipende proprio dall'utente specifico che lo utilizza!**

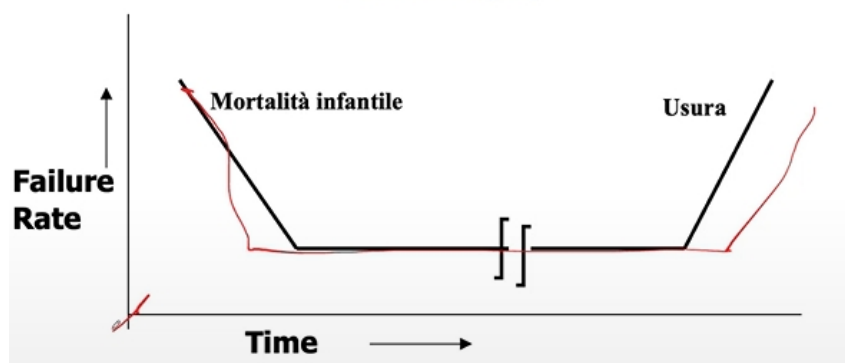
A questo punto possiamo confrontare l'affidabilità Hw e Sw:
mentre i guasti Hw sono dovuti al consumo e deterioramento dei componenti, alla loro rottura o al fatto che non si comportino più come specificato
i guasti Sw sono causati soltanto dalla presenza di difetti nei programmi in quanto il Sw non si consuma!

Per risolvere le problematiche Hw alla fine mi basta sostituire le componenti, mentre nel caso Sw i difetti sono latenti! Senza debug il sistema Sw continuerà sempre ad avere gli stessi difetti.

A causa di queste differenze le metriche usate per l'affidabilità Hw non sono estensibili al Sw.

Mentre nel caso Hw l'obiettivo dell'affidabilità è la stabilità (tenere cioè la frequenza di guasto costante, si vuole che ogni componente abbia almeno tot tempo di vita), per il Sw l'obiettivo dell'affidabilità è la far decrescere la frequenza di guasto.

Nella realtà: andamento frequenza di guasto hardware
(effetto dell'eliminazione dei componenti difettosi prima, e dell'usura poi)



Si inizia con alta probabilità di malfunzionamenti hardware (molti dispositivi hardware hanno malfunzionamenti all'inizio ad esempio a causa di difetti di fabbricazione, stress iniziali, danni da trasporto...) per poi avere un andamento costante.

Lez 3

Nel caso del software abbiamo visto che **l'affidabilità dipende dai difetti**.

Ci aspettiamo, riguardo l'andamento frequenza di guasto software, una curva con "mortalità infantile" proprio come nel caso hardware!

Quando un software viene rilasciato, questo contiene difetti. Nei primi mesi di rilascio gli utenti rilasceranno un feedback, che permetterà all'azienda di scorgerli ed eventualmente risolverli.

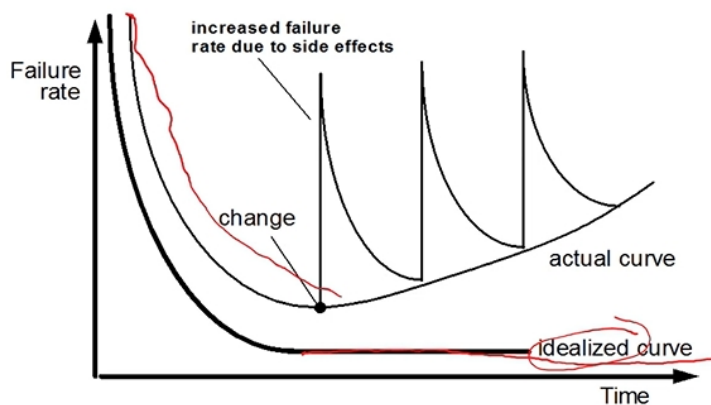
L'affidabilità migliora, e quindi la frequenza di guasto diminuisce.

Intuitivamente, la curva quindi dovrebbe partire con frequenza di guasto alta (mortalità infantile) e scendere nel tempo tendendo a frequenza di guasto zero all'infinito (idealized curve).

La realtà però è diversa: l'andamento del failure rate risale dopo un po' di tempo.

Andamento frequenza di guasto software

(effetto dell'eliminazione dei difetti prima, e dell'invecchiamento per manutenzione poi)



Si utilizza quindi il software fintanto che funziona, non appena si hanno malfunzionamenti di carattere bloccante lo mando in riparazione per poi riutilizzarlo. Questi blocchi del software a causa di problemi di affidabilità hanno comportato la definizione di una nuova metrica: **disponibilità del software**.

La disponibilità rappresenta la % di tempo che il sw è risultato utilizzabile nel corso della sua vita, e dipende dal numero di guasti che si verificano e dal tempo necessario per ripararli.

Tecnica per migliorare quindi la curva ed arrivare alla curva attuale del grafico: invece di "aspettare" che il software si guasti, viene definito un periodo (in base a un modello ben preciso) per cui il software dovrà essere "resettato" (**software rejuvenation**) e quindi tornare a uno stato più stabile. In questo modo l'indisponibilità del software viene ridotta.

È bene sottolineare che il software non si usura di per sé, ma dal momento che viene utilizzato nell'ambiente del sistema operativo (interagendo con molte altre componenti), è proprio quest'ultimo a contribuire a malfunzionamenti del software.

Nel tempo possono infatti accumularsi memory leak, frammentazione della memoria, gestione errata delle risorse che possono comportare a rallentamenti, instabilità fino a guasti improvvisi. Quindi software rejuvenation pulisce tutto per evitare questi guasti preventivamente.

Malfunzionamenti software comportano anche danni gravi, in quanto legato a sistemi di trasporto, traffico aereo, produzione e distribuzione di energia, comunicazione etc..

Una volta però introdotta la software rejuvenation, perché la curva risale comunque nel tempo? Sappiamo che la manutenzione nell'ingegneria del software è uno dei passi più importanti e costosi nella vita di un prodotto software.

Quando si effettua un passaggio per un software da una versione ad un'altra i cambiamenti possono essere molto significativi e ciò comporta nuovi difetti, ergo aumento della frequenza di guasto.

Per verificare l'affidabilità di un software si utilizza la tecnica di **Statistical Testing**. **Questo testing si basa sul prendere il prodotto, stabilire un periodo temporale per il testing. Il software viene testato quindi diverse volte, registrando ogni qual volta si osserva una failure del sistema.**

Si affiderà poi al modello di affidabilità questa sequenza di errori, che fornirà una stima di affidabilità in base al tempo di utilizzo ($p(t)$).

Se l'affidabilità non è sufficientemente alta per i propri obiettivi, si può procedere con l'investire per migliorare il software o rischiarsela.

Lez. 4 (12/10)

Processo Software

Rappresenta la serie di attività necessarie alla realizzazione del prodotto software nei tempi, costi e caratteristiche di qualità desiderate/pianificate.

Le attività in questione sono molteplici (es. fase di codifica) e sono partizionabili in due gruppi:

- **Fasi di Definizione** == si occupano di “**cosa**” il software deve fornire. In questa fase si definiscono i requisiti e si producono le specifiche. Non si sta costruendo, si sta cercando di capire cosa si vuole fare
- **Fasi di Produzione** == si sposta il focus dal cosa al “**come**” realizzare quanto ottenuto con le fasi di definizione. Si progetta quindi il software, si codifica, si integra e si rilascia al cliente

Queste due fasi vengono riconosciute principalmente nel primo dei tre stadi di ciclo di vita del prodotto software (sviluppo, manutenzione, dismissione) ma si possono ritrovare anche nello stadio di manutenzione dal momento che in tale fase si torna a fasi precedenti.

Si ricorda poi che durante ogni fase occorre fare testing di quanto prodotto mediante opportune tecniche di verifica (correttezza) e validazione (capire se ciò che ci è stato richiesto è stato correttamente realizzato).

Esistono diversi tipi di manutenzione:

- **Manutenzione Correttiva**: ha lo scopo di eliminare i difetti (fault) che producono guasti (failure) del software
- **Manutenzione Adattativa**: ha lo scopo di adattare il software a cambiamenti a cui è sottoposto l’ambiente operativo per cui è stato sviluppato (es. aggiornamento dell’hardware o di processi)
- **Manutenzione Perfettiva** (o evolutiva, è la più comune e molto costosa): ha lo scopo di estendere il software per accomodare funzionalità aggiuntive ed è costosa poiché in questo modo si allargano i requisiti del software stesso
- **Manutenzione Preventiva** (software reengineering, non è vista proprio come manutenzione ma attività propedeutica alla manutenzione): consiste nell’effettuare modifiche che rendano più semplici le correzioni, gli adattamenti e le migliorie. Si applica tipicamente per processi che non hanno documentazione recente (rischioso toccare il codice), in tal caso si parte dal codice per costruire il documento di progetto relativo al codice e da questo il documento di specifica, molto costoso.

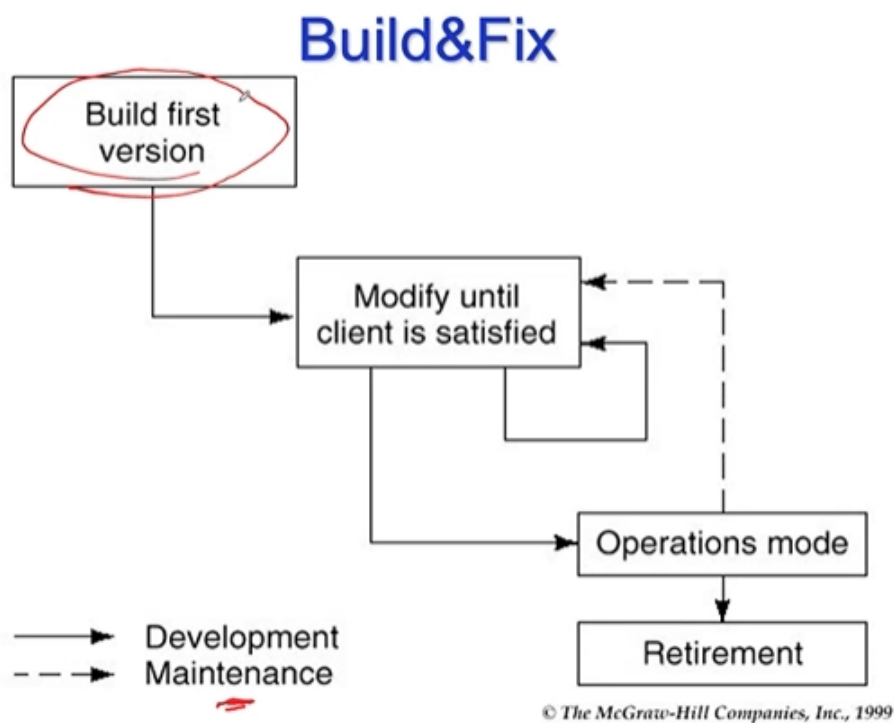
Abbiamo già definito il ciclo di vita del prodotto software come intervallo di tempo dalla nascita dell’esigenza di costruire un prodotto software all’istante in cui viene dismesso, ma esiste una definizione più dettagliata.

Il ciclo di vita del prodotto software include le fasi di definizione dei requisiti, specifica (specifica dei requisiti, analisi dettagliata), pianificazione, progetto preliminare, progetto dettagliato, codifica, integrazione, testing, uso, manutenzione e dismissione. *Si noti però che tali fasi possono sovrapporsi o essere eseguite in modo iterativo* (vedi TESTING! Che deve essere eseguito durante varie fasi).

Il **Modello di Ciclo di Vita** del software specifica come eseguire queste attività, ossia la serie di fasi attraverso cui il prodotto software progredisce e l'ordine in cui vanno eseguite dalla definizione dei requisiti fino alla dismissione.

Non esiste un modello migliore degli altri, dipende dalla natura dell'applicazione (tipo di software, a mercato, critico, a contratto...), dalla maturità dell'organizzazione, dai vincoli dettati dal cliente...

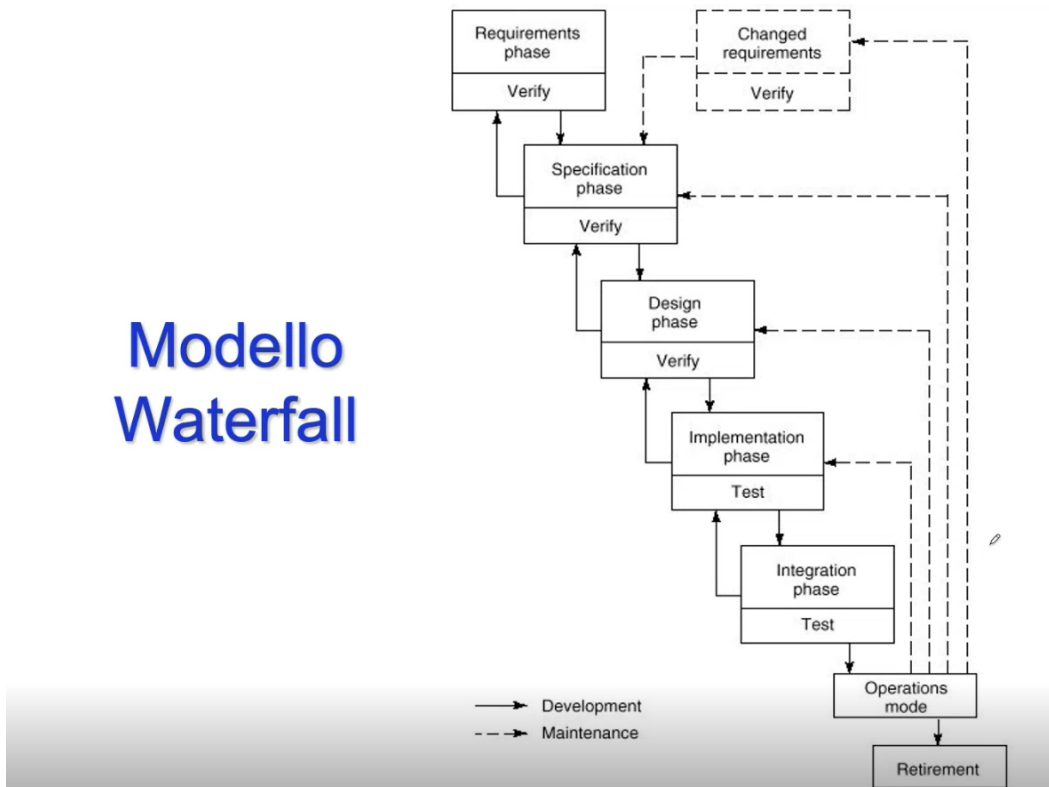
L'assenza di un modello corrisponde a una modalità di sviluppo detta **Build & Fix** o Fix it Later in cui il software viene creato e modificato finché il cliente è soddisfatto. Visivamente ogni rettangolo attività



“**crisi del software**” se si itera continuamente nell’attività di modifica finché il cliente non è soddisfatto.

Un primo modello nato nel 1970, due anni dopo la nascita dell’isw, è il **Modello Waterfall**.

Modello Waterfall



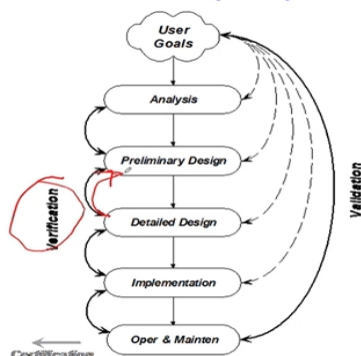
Secondo questo modello a differenza del primo le attività devono esser svolte in rigida sequenza. Es. passo alla fase di specifica solo quando la fase dei requisiti è stata completata e VERIFICATA.

Nei blocchetti sotto troviamo Test invece che Verify, ciò dipende dagli artefatti: i primi tre blocchi in realtà dovrebbero prevedere il termine Static Verification piuttosto che Verify poiché si verificano manualmente mentre gli ultimi 3 Dynamic Verification.

Questo modello assume che una volta finita la parte di specifica non tornerò più a toccare i requisiti durante lo sviluppo ma ciò è irrealistico (es. modifiche normative). Un'altra limitazione è nell'assenza di feed da parte del cliente, che viene coinvolto in fase iniziale ma non nelle fasi tecniche e si rischia quindi di arrivare ad un prodotto che non soddisfa le esigenze del cliente.

I modelli successivi cercano di sfruttare i vantaggi e andare oltre gli svantaggi del modello waterfall.

Verification & Validation (V&V) nel Waterfall

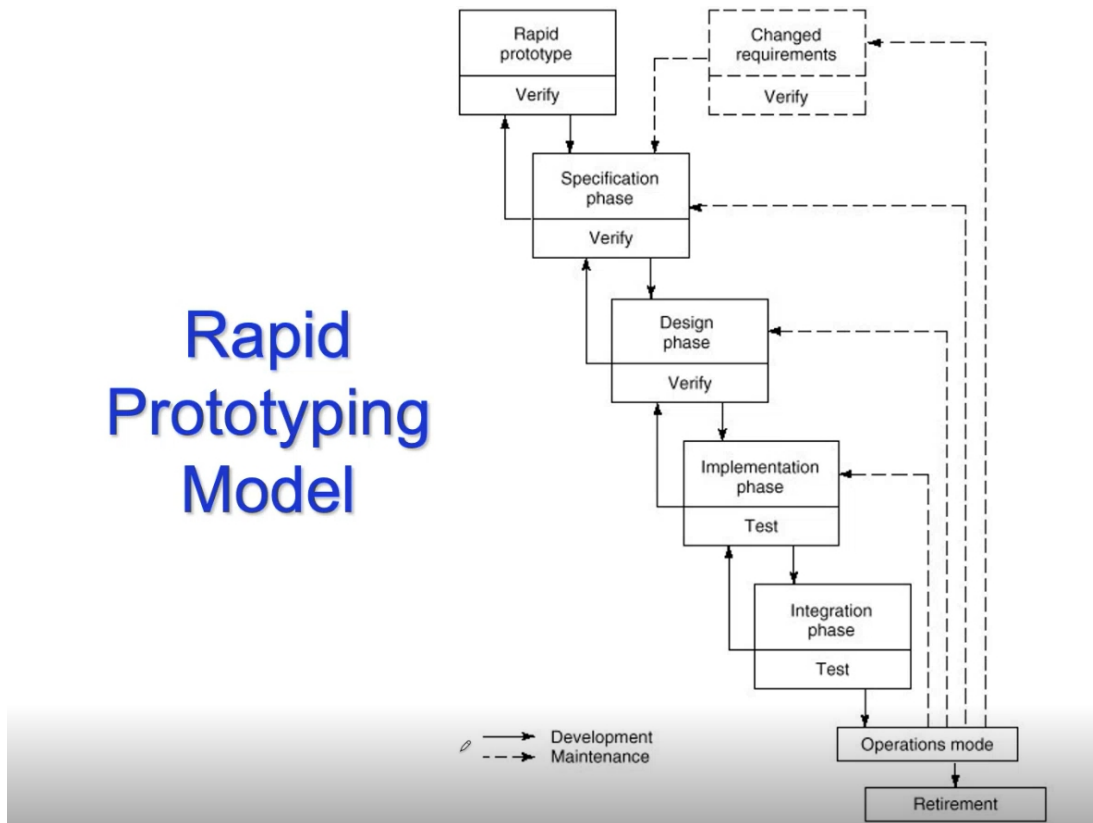


Prima di passare agli altri modelli distinguiamo verifica e validazione: **Verifica** per controllare che il progetto che ho prodotto dopo una certa fase è corretto rispetto al documento ricevuto in input.

Ciò non basta perché potrebbe essere corretto ma non valido: ad ogni fase dovrò quindi verificare che ciò che ho prodotto soddisfi gli obiettivi dell'utente (ossia ad ogni fase è importante verificare che sto rispecchiando gli obiettivi iniziali). (tratteggiate sopra e continua sotto per ricordare che farlo alla fine è più facile ma va fatta sempre).

In basso a sinistra **Certificazione:** è la dichiarazione formale relativa alla qualità del prodotto. (es. DOCG: esiste uno standard di riferimento e certifico che il mio prodotto soddisfa lo standard). *Nel caso dei prodotti software non esiste lo standard diretto quanto delle certificazioni della capacità di un'organizzazione dell'utilizzo delle tecniche migliori* (maturità dell'organizzazione vista prima).

Il modello a **Prototipo Rapido** è stato il primo modello migliorativo post waterfall.

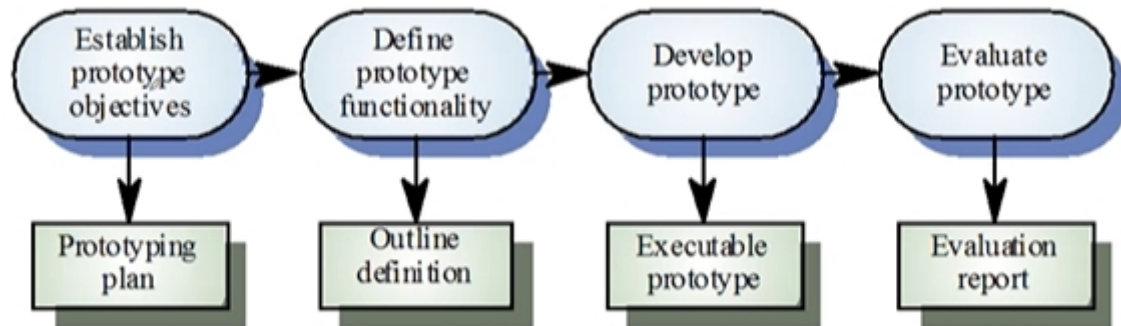


La differenza rispetto a waterfall è come viene svolta la prima fase, quella più “alta” che qui è chiamata Rapid Prototype. **Si cerca di superare la limitazione per cui in waterfall vi potrebbero essere ambiguità nella fase dei requisiti nella comunicazione con il cliente che poi una volta in fase di sviluppo non sarà più in grado di intervenire.** Il prototipo nel caso del software rappresenta soltanto *l'implementazione dell'interfaccia del prodotto*. Ciò è importante in quanto risolve uno dei principali problemi in fase di requisiti: un utente non sa descriverti come vorrebbe l'interfaccia grafica, gliela devi fa vede.

Questo prototipo deve essere rapido: devo poterlo mettere a disposizione nell'arco di massimo una settimana/10 giorni.

Quindi questo modello ha l'obiettivo di tirar fuori i requisiti (**Requirements Elicitation**, l'utente può sperimentare con il prototipo per vedere come il sistema supporta i loro bisogni) e dall'altra validarli (**Requirements Validation**, il prototipo potrebbe rivelare errori o omissioni di requisiti).

Riduzione quindi del rischio!



Si segue questo procedimento per produrre un prototipo (curve attività, quadrati documenti prodotti).

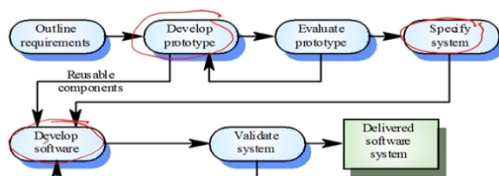
A partire da questo modello si è pensato di usare il prototipo anche come strumento di specifica, ossia come qualcosa da cui partire per poi sviluppare piano piano il progetto.

Ciò tuttavia deve essere trattato con attenzione in quanto *requisiti non funzionali non possono essere testati adeguatamente via prototipo* (es. tempo di risposta quando metto username e password), inoltre l'implementazione non ha valenza legale nel contratto e *alcuni requisiti come funzioni critiche di sicurezza potrebbero essere impossibili da soddisfare per un semplice prototipo*.

Quindi si vuole che il prototipo che si produce sia rapido e Usa e Getta, ossia che venga utilizzato solo in funzione della fase dei requisiti per poi iniziare a parte il vero e proprio progetto secondo il modello Waterfall. Quindi il prototipo non dovrebbe mai essere considerato come prodotto finale

Alcuni dei componenti del prototipo potranno essere in alcuni casi utilizzati per la realizzazione dell'interfaccia grafica finale.

Throw-away prototyping process



Lez 5 (16/10)

Principale svantaggio Rapid Prototype Model: tipicamente questo prototipo è usa e getta ed è **difficile convincere l'utente della difficoltà della realizzazione del progetto effettivo, ciò può mettere pressione al team di sviluppo.**

Per garantire che il prototipo venga realizzato rapidamente si utilizzano le tecniche di Visual Programming (programmazione visuale) che mettono a disposizione una grande libreria di componenti preconfezionate tipiche per la realizzazione di interfacce grafiche (es. Visual Basic).

Importante far capire al cliente che il progetto vero e proprio non può basarsi solo su questa tecnologia ma è molto più complesso e richiede molto più tempo.

Infatti lo sviluppo visuale non è adatto per essere utilizzato in situazioni basate su lavoro di gruppo, inoltre non esiste un codice vero e proprio dietro la sua generazione pertanto non può essere ottimizzato e ciò causerebbe inoltre problemi di mantenimento (il codice è difficile da modificare in quanto autoprodotta).

Con il tempo, invece di modificare l'approccio Waterfall per migliorarlo, si sono introdotti approcci che rompersero la struttura rigida e monolitica del modello in favore di altro. Tali approcci fanno uso della **Process Iteration**: *piuttosto che sviluppare il prodotto in modo monolitico come sequenza di macrofasi si è pensato di introdurre iterazioni che lavorassero su parti del prodotto più piccole.*

L'introduzione di tali approcci deriva ancora una volta dal problema principale di Waterfall: è irrealistico pensare che nell'arco dello sviluppo del progetto effettivo non si vadano più a toccare i requisiti.

Due approcci: **Sviluppo Incrementale** e **Sviluppo a Spirale**.

L'idea dello **sviluppo incrementale** è *sviluppare e consegnare il prodotto in incrementi successivi dopo aver stabilito una architettura d'insieme.*

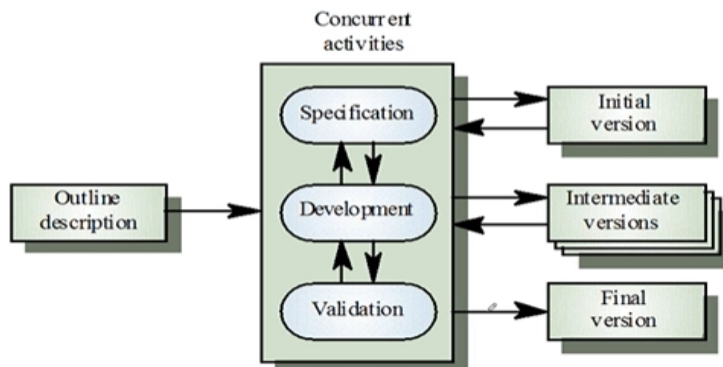
Si ricordi la relazione quadratica che esiste tra costo e dimensione del software: il costo per realizzare un software di dimensione S è doppio rispetto a realizzare due software di dimensione $S/2$ -> **è conveniente sviluppare il prodotto software dividendolo in più incrementi** (tuttavia si ricordi che poi rimettere i pezzi insieme non ha costo zero).

Oltre ai costi un altro vantaggio sta nel fatto che il cliente vede continuamente i progressi dello sviluppo: periodicamente il cliente riceve gli incrementi che estendono quanto gli è già stato consegnato prima (l'ultimo incremento consegnato corrisponde alla consegna finale del prodotto).

Inoltre non si perdono nemmeno i benefici Waterfall dal momento che ciò viene fatto con un approccio rigoroso e strutturato.

Modello incrementale

- Il prodotto software viene sviluppato e rilasciato per incrementi (*build*) successivi

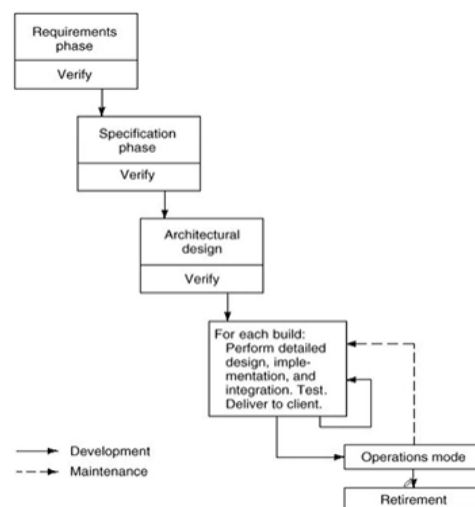


Si parte da una **outline description** (descrizione d'insieme) del prodotto e **ogni attività può essere svolta in modo concorrente da team separati risparmiando tempi di sviluppo**. La prima build sarà l'Initial Version (molto limitata), si prosegue con le successive fino alla finale.

L'approccio Incrementale può essere realizzato in due versioni alternative:

- con **Overall Architecture**

Versione con overall architecture



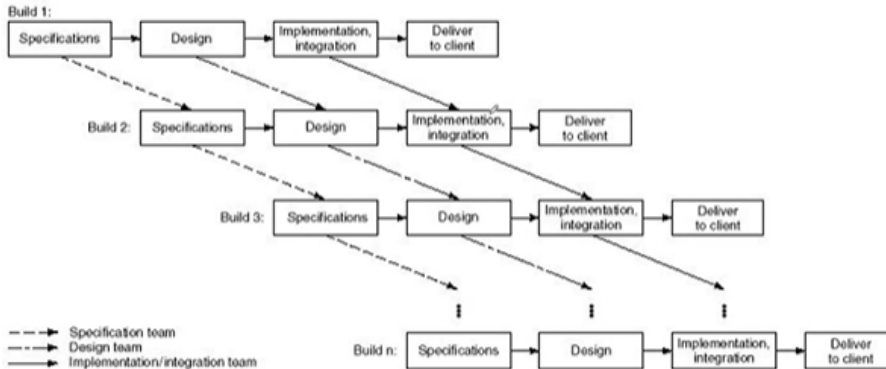
È la versione più “conservativa” e vicina al modello Waterfall in quanto è identica fino alla progettazione architetturale per poi procedere a partire dalla fase di progettazione dettagliata in modo incrementale.

(ogni fase di progettazione è divisa in architetturale o preliminare e di dettaglio. Nella prima si progetta l'architettura del software, individuando le componenti principali del prodotto software e le relazioni che esistono tra loro. Dopo averle individuate, questa versione considera ciascuna componente come se fosse una build procedendo quindi con l'approccio incrementale).

Dopodiché si va in **modalità operativa** (manutenzione finché il prodotto non viene dismesso).

- **Senza Overall Architecture** (più rischiosa)

Versione senza overall architecture



I build non vengono identificati a partire dall'architettura software ma a partire dai requisiti di alto livello del software.

Es. Sto sviluppando un simil-Word, devo implementare 50 (n) funzionalità. Parto da quella con priorità più alta (Build 1) e lavoro come fosse waterfall, poi passo alla seconda fino all'n-esima.

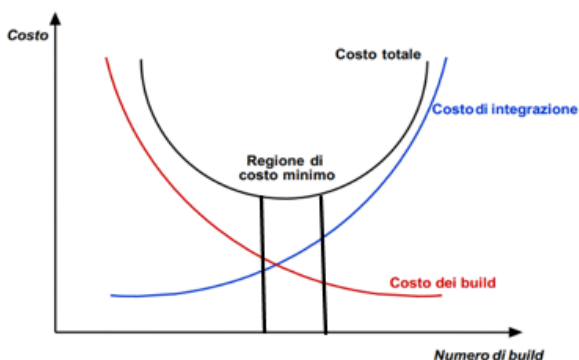
Anche in questo caso se si avessero a disposizione un team per ogni build potrei farli lavorare in parallelo e all'inizio vado sicuramente più veloce rispetto all'altra versione, tuttavia diversi rischi!

In particolare nella parte Integration nel grafico, non posso consegnare ogni build separatamente come fossero applicazioni separate, ogni build deve essere integrata con quelle precedenti. **Con l'architettura software sapevo esattamente quali componenti interagissero tra loro e come integrarli di conseguenza.**

Senza architettura non ho queste informazioni ed è possibile che alla k-esima build scopro problemi di integrazione con build che avevo già consegnato.

A questo punto per l'approccio incrementale si deve sapere quale sia il numero di build più conveniente da adottare in base al caso. Il criterio su cui questa scelta si basa è **l'impatto sui costi**.

Impatto sui costi del software



Maggiore il numero di build minore il costo per la relazione vista prima, ma c'è da considerare il costo d'integrazione che ha andamento opposto!

Si identifica una regione di costo minimo dove conviene scegliere il numero di build da utilizzare.

Facciamo un confronto tra Waterfall e Incrementale.

Nel modello Waterfall è più difficile accomodare modifiche ai requisiti, mentre in quello incrementale requisiti suddivisi in classi di priorità e facilmente modificabili (modificare un requisito che ha impatto sul singolo build significa lavorare su una più piccola parte di codice).

Nel primo si può avere feedback dal cliente solo una volta terminato lo sviluppo, nel secondo continuo feedback dal momento che gli mando continuamente roba.

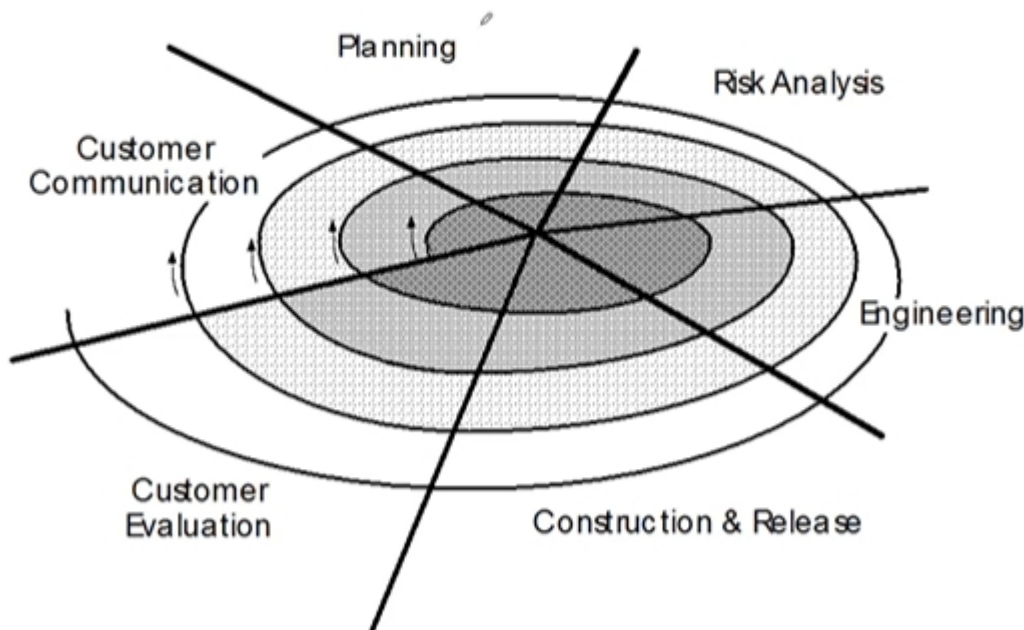
Nel primo le fasi sono condotte in rigida sequenza (l'output di una costituisce l'input per la successiva e si passa alla prossima solo dopo verifica), nel secondo le fasi possono essere condotte in parallelo.

Nel primo tutte le fasi sono sviluppate sull'intero prodotto mentre nel modello incrementale, anche considerando la versione con overall architecture a partire dalla fase di progettazione dettagliata il resto delle fasi sono effettuate sul singolo build.
Nel primo caso il team di sviluppo è costituito da un gran numero di persone, nel secondo posso avere diversi team di sviluppo, ciascuno con piccole dimensioni.

Dall'approccio incrementale si è derivato l'approccio **Agile**. Ancora oggi alcuni progetti basati su Waterfall se vincolo esterno (es. cliente).

Il secondo modello che avevamo anticipato era il **Modello a Spirale**.

Modello a spirale



Si utilizza questa rappresentazione per dare idea sia del tempo che dei costi.

Si parte dalla spirale più piccola andando sempre in esterno. **La dimensione radiale rappresenta l'incremento dei costi, quella angolare l'avanzamento del tempo** (dalla spirale più interna all'inizio fino a quella più esterna alla fine).

Ad ogni cerchio della spirale faccio sempre le stesse attività iterativamente.

Si divide il piano in vari settori, si parte da dove stanno le frecce iniziando quindi con la Customer Communication, poi si pianificano le attività da svolgere per poi arrivare a un settore che è caratteristica peculiare di questo modello: la **Risk Analysis**.

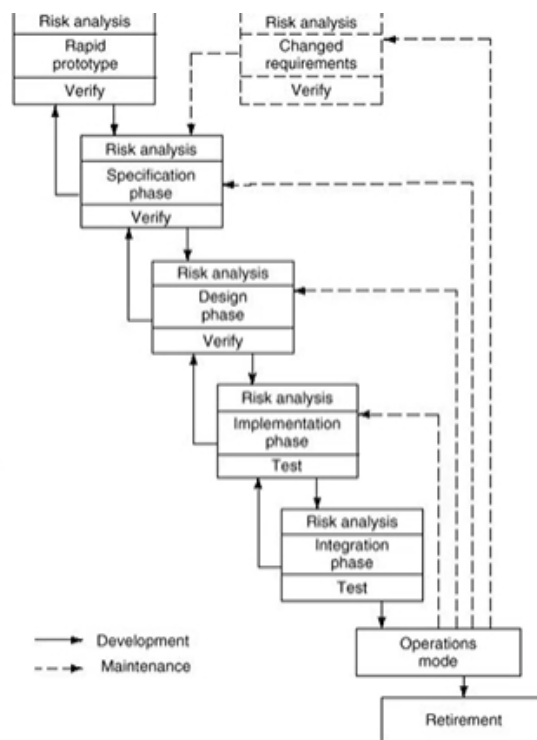
Se questa analisi porta a una valutazione secondo la quale i rischi sono eccessivi allora potrebbe essere conveniente valutare l'idea di fermare il progetto (e questo vale per ogni iterazione, anche ad es. alla decima spirale).

Se conviene procedere si passa all'Ingegnerizzazione (specifiche, progettazione etc.), poi si fa Costruzione e Rilascio (codifica, costruzione e rilascio) e infine Customer Evaluation (si consegna al cliente quella che può essere una build) per poi ripartire.

Ogni cerchio della spirale corrisponde quindi a una sorta di build.

Si considererà poi allo stesso modo la manutenzione come un aggiuntivo cerchio della spirale.

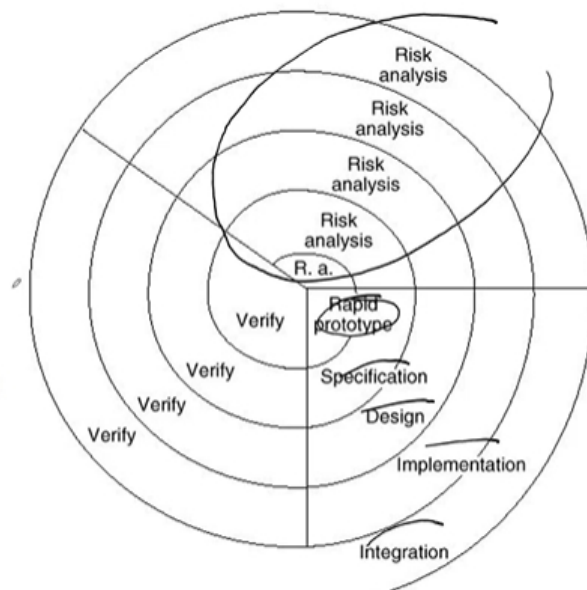
**Modello
a spirale
semplificato
(versione
lineare)**



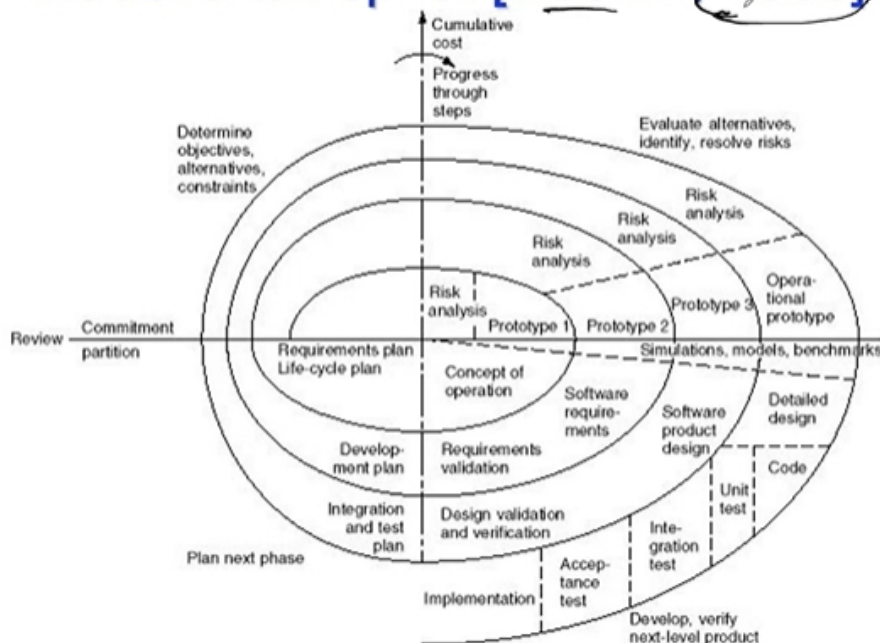
Esiste anche questa rappresentazione del modello a spirale semplificata che è utile per mostrare la differenza rispetto al Waterfall: ad ogni blocco viene aggiunta inizialmente una fase di risk analysis che deve essere effettuata prima di passare alla successiva.

Da qui il corrispettivo a spirale:

Modello a spirale semplificato



Modello full-spiral [Boehm, 1988]



Sopra il primo modello a spirale pubblicato, Boehm ha dimostrato con risultati su diversi progetti della Nasa che **usando il modello a spirale si ottiene un miglioramento della produttività pari a circa il 50%.**

Anche qui centrale la fase di analisi dei rischi, dove vediamo nella spirale dei Prototipi. Questi sono ovviamente diversi da quanto visto nel modello a prototipo rapido, sono infatti prototipi creati appositamente per facilitare l'analisi dei rischi. Nella fasi di Develop in basso a destra invece si fa uso di simulazioni, test e benchmark per aiutarsi (questo perché si faceva riferimento a software scientifici Nasa commissionati dalla nasa per la nasa, non a contratto).

Qui entra in gioco la caratteristica importante del modello.

Un modello simile non ha senso per software a contratto!

Es. Arrivare a un mese dalla consegna e dire al cliente che l'analisi dei rischi suggerisce che l'azienda potrebbe risentirne non è d'interesse del cliente, e inoltre se l'azienda non consegna il progetto completo come concordato a causa di eventuali rischi nascono anche questioni legali.

Questo modello è quindi sicuramente molto efficace, ma si è applicato con successo **solo per software di tipo interno.**

Un'altra caratteristica è che si devono avere persone molto competenti per l'attività di analisi di rischi.

In realtà questa attività viene ovviamente fatta in un qualsiasi progetto software, ma nel modello a spirale diventa un elemento chiave.

Ma in generale cosa significa concretamente fare analisi dei rischi?

Risk Management

In ogni tipo di progetto, non solo software, viene realizzata l'attività di risk management. **Si tratta della serie di strumenti e tecniche per identificare e minimizzare i rischi.**

Un rischio è definito come la probabilità che possa capitare una circostanza avversa nel corso dello sviluppo del software.

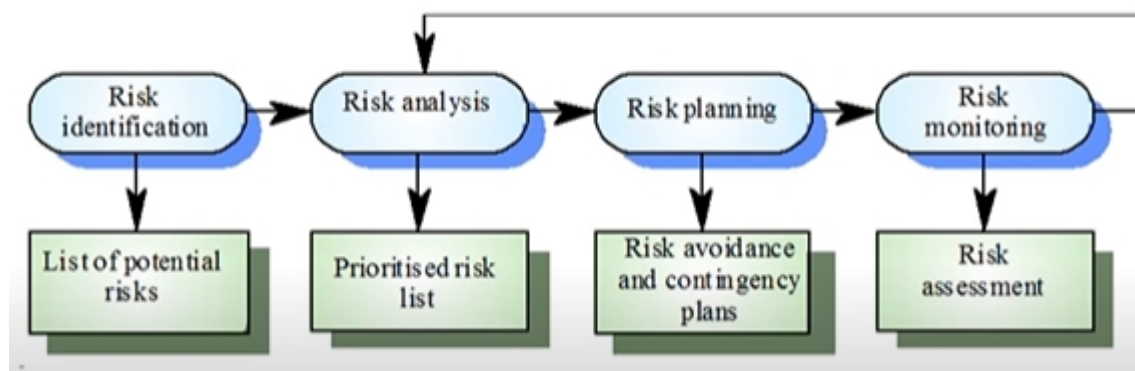
Esistono diversi tipi di rischio in base al loro effetto: **Project Risks** == effetti su tempo e risorse, **Product Risks** == effetti su qualità e performance del software in produzione, **Business Risks** == effetti sull'organizzazione che sviluppa il software

Es. Staff Turnover per cui staff specializzato lascia il progetto prima che finisca, è un Project Risk. Requirements Change (modifica di più requisiti di quanto ci si aspettasse) è un rischio di tipo Project e Product.

Size Underestimate è rischio di tipo Project e Product e rappresenta il fatto che si abbia sottostimato la dimensione del sistema (quindi ho messo ad esempio meno risorse di quante ne sono necessarie). Product Competition e Technology Change come Business Risk.

Bisogna considerare quindi tutto ciò che potrebbe avere impatto negativo sul progetto e la probabilità che accada.

Il risk management come detto quindi rappresenta un sottoprocesso nel processo software, e in quanto tale presenta diverse attività in sequenza.



(ancora cerchi attività rettangoli output attività)

Si parte con la **Risk Identification** dove si avrà in output un documento contenente tutti i possibili rischi. Si procede con la **Risk Analysis** (come visto focale nel modello a Spirale) dove l'output è una lista dei rischi a cui è data una priorità. Fare risk analysis infatti significa identificare non solo la probabilità del rischio ma anche i suoi effetti in modo da identificare la priorità.

Si passa alla **Risk Planning** dove ci si concentra sui rischi di priorità maggiore e si pensa a delle strategie per poter reagire e si arriva al **Risk Monitoring**.

Mentre le prime tre sono fatte in sequenza il Risk Monitoring è fatto periodicamente durante lo sviluppo per rianalizzare eventuali rischi che possono essere cambiati, possono esserci fattori di rischio che mi indicano che sono campanelli di allarme e in generale si cerca di capire se si devono rianalizzare certi rischi.

Lez 6 (19/10)

Vediamo più precisamente le varie fasi.

Risk Identification: tra le tipologie di rischio maggiormente caratteristici nel caso di prodotti software si hanno rischi legati alla **tecnologia**, alle **persone**, **all'organizzazione**, **tool di supporto** (es. IDE), **requisiti**, **stima** (stimare la durata di un prodotto software è molto importante specie a contratto). Vediamo un esempio.

Risk identification (2)

Risk type	Possible risks
Technology	The database used in the system cannot process as many transactions per second as expected. Software components which should be reused contain defects which limit their functionality.
People	It is impossible to recruit staff with the skills required. Key staff are ill and unavailable at critical times. Required training for staff is not available.
Organisational	The organisation is restructured so that different management are responsible for the project. Organisational financial problems force reductions in the project budget.
Tools	The code generated by CASE tools is inefficient. CASE tools cannot be integrated.
Requirements	Changes to requirements which require major design rework are proposed. Customers fail to understand the impact of requirements changes.
Estimation	The time required to develop the software is underestimated. The rate of defect repair is underestimated. The size of the software is underestimated.

Si passa alla **Risk Analysis**. Si deve, per ogni rischio identificato, associare la probabilità di occorrenza del rischio e l'effetto che avrebbe. Le probabilità non sono ovviamente calcolate precisamente ma disposte secondo **range di valori** (very low < 10%, low 10-25%, moderate, high, very high) mentre gli effetti possono essere **catastrofici** (il progetto fallisce se succede), **seri**, **tollerabili** o **insignificanti**.

Risk analysis (2)

Risk	Probability	Effects
Organisational financial problems force reductions in the project budget.	Low	Catastrophic
It is impossible to recruit staff with the skills required for the project.	High	Catastrophic
Key staff are ill at critical times in the project.	Moderate	Serious
Software components which should be reused contain defects which limit their functionality.	Moderate	Serious

Si considerano poi i rischi come alta priorità quei rischi di effetto catastrofico o di effetto serio ma con probabilità almeno alta. Si fa poi l'identificazione dei top-ten risks classificandoli in ordine di importanza.

Si passa alla **Risk Planning**. Si hanno due tipi di strategie: **Avoidance Strategies** con obiettivo di ridurre la probabilità di occorrenza del rischio, **Minimisation Strategies** con obiettivo di ridurre l'effetto del rischio sul progetto o prodotto. Sarebbe ideale applicare entrambe le strategie ad ogni rischio ma ogni rischio è diverso e per alcuni è conveniente usare una strategia piuttosto che un'altra.

Es. rischio che il capo degli analisti software se ne vada durante lo sviluppo del progetto, Avoidance Strategy gli aumento lo stipendio per ridurre la probabilità che se ne vada, invece la Minimisation Strategy è più difficile da applicare in quanto una figura del genere è sicuramente difficile da sostituire nel breve tempo soprattutto se il progetto è già in fase di sviluppo. Al contrario col rischio che si ammalino molti membri del team, non posso avoidance ma posso minimizzazione.

Nel caso in cui per un certo rischio non si possa adottare nessuna delle due strategie, si adottano **Contingency Plans** (piani di contingenza, "piani B").

Risk management strategies

Risk	Strategy
Organisational financial problems	Prepare a briefing document for senior management showing how the project is making a very important contribution to the goals of the business.
Recruitment problems	Alert customer of potential difficulties and the possibility of delays, investigate buying-in components.
Staff illness	Reorganise team so that there is more overlap of work and people therefore understand each other's jobs.
Defective components	Replace potentially defective components with bought-in components of known reliability.

Infine, periodicamente, occorre fare **Risk Monitoring**. Si deve controllare periodicamente se qualcosa è cambiato e ciò si fa normalmente in riunioni manageriali per controllare l'avanzamento del progetto.

Però esistono anche dei **fattori di rischio**, che rappresentano dei *campanelli d'allarme in quanto fanno capire come alcuni rischi non sono stati considerati o l'avevo considerato in modo diverso rispetto a come dovevo considerarlo* (in caso di fattori di rischio si deve essere in grado di attivarsi immediatamente).

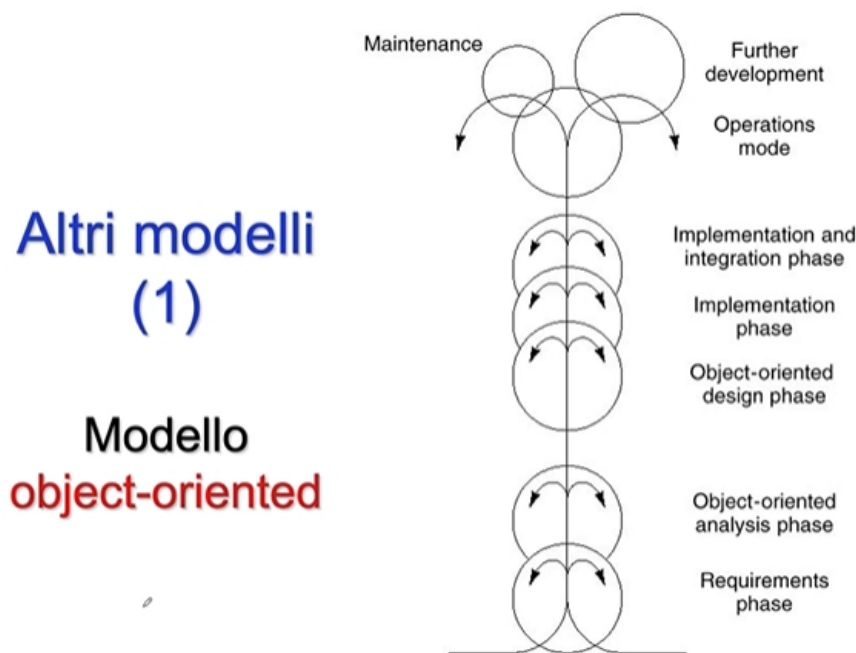
Tra fattori di rischio ad esempio se vedo che la gente non ha voglia di fare e quello potrebbe essere un fattore di rischio a livello di produttività.

Risk monitoring (2)

Risk factors

Risk type	Potential indicators
Technology	Late delivery of hardware or support software, many reported technology problems
People	Poor staff <u>morale</u> , poor relationships amongst team member, job availability
Organisational	organisational <u>gossip</u> , lack of action by senior management

Vediamo infine altri modelli di processo software che vale la pena citare.



Il **modello Object Oriented** o **a Fontana** prevede un approccio object oriented all'analisi dei requisiti e alla progettazione del software.

Abbiamo visto java come OOP, *l'idea è di applicare lo stesso paradigma non solo alla codifica del software ma anche all'analisi dei requisiti e alla progettazione del software*. Infatti guardando la figura l'etichetta object-oriented sta soltanto sulla fase di analisi dei requisiti e sulla fase di progettazione. Infatti **un software progettato secondo questo approccio potrebbe essere implementato anche facendo uso di un linguaggio di programmazione non object oriented**.

Ciò che differenzia questo modello dal Waterfall è **la possibilità per ogni fase di essere sviluppata in modo concorrente e iterativo**, la concorrenza viene messa in evidenza in quanto le fasi (i cerchi) non sono distinte l'una dall'altra ma si sovrappongono, ossia ad esempio mentre faccio la fase di definizione requisiti posso già iniziare anche l'analisi dei requisiti (mentre in waterfall sarebbe stato in rigida sequenza). L'iterazione viene invece messa in evidenza dalle frecce: quelle all'interno dei cerchi rappresentano le iterazioni intra-fase (all'interno di una singola fase, es. raffino i requisiti) e quelle che escono dal cerchio che invece sono inter-fase (posso tornare indietro).

Poi c'è la manutenzione come cerchio piccolo in alto a sinistra, è più piccolo perché si mette in evidenza il fatto che usando questo approccio il software prodotto sarà più facile da modificare e quindi più semplice manutenzione.

Un altro modello è quello di **Ingegneria Simultanea o Concorrente**.

L'idea è ridurre i costi e i tempi di sviluppo tramite un approccio concorrente allo sviluppo del prodotto e del processo ad esso associato. Quindi le fasi di sviluppo coesistono invece di essere eseguite in sequenza.

Questo approccio è molto efficace, ma **si devono utilizzare strumenti software che**

supportino l'utilizzo di questi approcci, che permettano di gestire in modo molto efficace il progetto dal momento che se ne svolgono diverse parti in parallelo.

Tra questi strumenti tipicamente si ha un **information sharing** e un **componente di project management che permetta di gestire in modo efficace tutte le attività**.

Si fanno delle riunioni online invece che dal vivo (dato che per natura del modello si lavora in parallelo è probabile che team diversi si trovino molto lontani) condividendo file e informazioni.

Di tutt'altra natura è il modello basato su **Metodi Formali**. **Questo modello è utile solo per particolari tipi di software: i software critici.**

La caratteristica di questo approccio è che l'analisi dei requisiti deve essere realizzata usando tecniche e linguaggi di specifica formali. *Un linguaggio di questo tipo è matematico e permette di specificare il software in modo non ambiguo e ha il vantaggio di permettere l'utilizzo di tecniche di verifica automatizzate.*

Un esempio di realizzazione di questo modello è la cleanroom software engineering (ing. soft. in camera sterile) il cui nome suggerisce come già a partire dall'analisi dei requisiti per questi progetti si debba fare molta attenzione a non inserire errori.

La maggioranza dei modelli visti si applicano con successo per software a contratto (cliente che commissiona).

Ci spostiamo ora nello scenario in cui invece il **software è realizzato dall'organizzazione tramite investimenti per il mercato**. Vedremo due modelli di cicli di vita organizzati da due organizzazioni differenti: *la prima che sviluppa software di tipo commerciale* (pubblica chiedendo ai clienti di pagare la licenza d'uso) e la seconda che invece *sviluppa software di tipo freeware*.

Nel primo caso guardiamo **Microsoft** per cercare di capire come satana produce il proprio software.

All'interno dell'azienda, a quanto ci dicono **Selby e Cusumano** (due ricercatori che sono stati ospitati 2 anni e hanno pubblicato un articolo dove spiegavano roba), si adotta un approccio iterativo, incrementale e concorrente per rispondere ai problemi di incremento della qualità dei prodotti software e riduzione di tempi e costi di sviluppo che aveva avuto fino a metà anni '80.

A quanto pare c'era anche particolare focus *nell'esaltare le doti di creatività delle persone coinvolte* nello sviluppo di prodotti software.

L'approccio attualmente adottato da Microsoft è noto come "**Synchronize-And-Stabilize**" ed è basato su:

- **Sincronizzazione quotidiana** (una volta al giorno) delle attività svolte da persone che lavorano sia individualmente che in piccoli team (3/8 persone), *assemblando quanto viene da loro fatto creando così una build (daily build) da testare ed eventualmente correggere.*

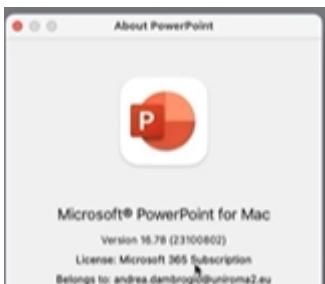
In pratica quando lo sviluppatore arriva in ufficio si collega ad uno strumento di **configuration management (repository** centralizzato del codice) e in base all'utente scarica sulla macchina la parte di prodotto su cui dovrà lavorare quella giornata. Lo sviluppatore sarà libero di realizzare quanto richiesto come preferisce, l'unico

vincolo è che dovrà inserire in repository quanto fatto entro una certa ora.
Il sistema raccoglie tutto, compila, e crea un eseguibile (daily build).

Chiaramente non si riesce tipicamente a realizzare subito una build funzionante, l'idea di questi figli di puttana è che chi ha realizzato la parte di codice che non va bene si dovrà fermare oltre tempo stabilito per sistemare.

- **Stabilizzazione**: periodicamente viene fatta un'attività di stabilizzazione in cui avviene un incremento significativo del prodotto (milestone), tipicamente 3/4 volte nell'arco dell'intera realizzazione del software (si correggono bug, si integra..)

Lez 7 (23/10)



23100802 rappresenta il numero dell'ultima daily build dell'applicazione.
Per quel che riguarda invece il ciclo di sviluppo del software in Microsoft questo avverrebbe in tre fasi: **Fase di Planning** (più a carattere tecnico-gestionale in cui si pianificano le attività di sviluppo), **Fase di Development** (si procede concretamente alla realizzazione di quanto pianificato) e **Fase di Stabilization** (da non confondere con la stabilizzazione periodica, è la fase che precede l'immisione del mercato e si fa testing interno ed esterno, si realizza la versione finale del prodotto).

Vediamole più nel dettaglio.

- **Pianificazione**: si procede anzitutto generando la **product vision**, si parte quindi con la **Vision Statement** ossia una breve descrizione che articola la visione che c'è dietro il prodotto (*necessaria perché non ho un cliente che mi chiede il software ma io azienda lo voglio fare*). A questa attività lavorano i **Product e Program Managers** (il primo esperto più che altro di marketing, i secondi sono coloro che stanno a capo di ciascun team. Questi ultimi sono poi coordinati da una terza figura, **il Project Manager**, che è responsabile tecnico dell'intero progetto).

Si procede quindi a partire da quanto fatto con la **Vision** alla realizzazione di un **Documento di Specifica**, che è più tecnico e rappresenta proprio un documento di specifica dei requisiti. Qui non lavora più il Product Manager. Infine si ha l'attività di **Schedule and Feature Team Formation**, ossia **basandosi sul documento di specifica si pianificano nel concreto le attività formando i vari team**. Ognuno costituito approssimativamente da un Program Manager, 3-8 sviluppatori e ad ogni sviluppatore è associato un tester (tipicamente tester e sviluppatore hanno le stesse competenze, quindi uno può fare tester ma anche sviluppatore in team diversi).

- **Sviluppo**: è una fase che procede per milestones. Si definiscono infatti un numero limitato di sottoprogetti, nell'esempio tre, la fine di ognuno di essi rappresenta una milestone e quindi un punto di stabilizzazione. Nel concreto comunque si progetta

codifica e debugga il codice. I tester insieme agli sviluppatori per testing continuo. In accordo a quanto visto nel modello incrementale il primo sottoprogetto riguarda le funzionalità più critiche.

- **Stabilizzazione**: si effettua **testing interno/alpha testing** == *testing effettuato all'interno dell'azienda del prodotto*, e **testing esterno/beta testing** == *testing effettuato da parte di user esterni (utenti selezionati, OEMs i produttori di hardware, ISVs gli sviluppatori di software) al di fuori dell'azienda. Dopodiché si arriva alla versione finale per poi distribuirla.*

Selby e Cusumano hanno infine identificato *alcune delle strategie messe in atto da Microsoft durante lo sviluppo software.*

La **prima strategia** è dedicata a definire prodotto e processo e **consiste in** **“considerare la creatività come elemento essenziale”** e prevede 5 principi di realizzazione: a) dividere il progetto in milestone, tipicamente 3 o 4, b) definire una product vision e una specifica funzionale che non è congelata come in waterfall ma evolverà durante il progetto, c) selezionare funzionalità e relative priorità in base alle necessità utente (*dove l'utente è il Product Manager, non esiste cliente poiché non è software a contratto*), d) definire un'architettura modulare per replicare nel progetto la struttura del prodotto (dal momento che si lavora con tanti team bisogna decomporre modularmente il prodotto per distribuire i compiti) e infine e) assegnare task elementari e limitare le risorse (Microsoft ha persone molto competenti e ciò è vantaggioso in generale ma svantaggioso perché si tratta di persone incontrollabili, se gli si impongono degli standard se ne vanno).

Per controllarle indirettamente si assegnano quindi dei task molto elementari senza dirgli esattamente come risolverli ma limito le risorse ossia gli dico di farlo entro un certo tempo).

La **seconda strategia** riguarda lo sviluppo e la consegna dei prodotti e **consiste nel** **“lavorare in parallelo con frequenti sincronizzazioni”**. a) si vuole sincronizzare il lavoro fatto dai vari team di sviluppo mediante la produzione del daily build, b) avere sempre un prodotto da consegnare con versioni per ogni piattaforma e mercato, c) usare lo stesso linguaggio di programmazione all'interno dello stesso sito di sviluppo, d) testare continuamente il prodotto ed e) usare metriche per il supporto delle decisioni (*in realtà quest'ultimo principio è usato in modo molto limitato perché non essendovi un contratto da rispettare a tutti i costi l'attenzione risiede più che altro nella promessa che l'azienda fa al mercato riguardo il tempo di consegna*).

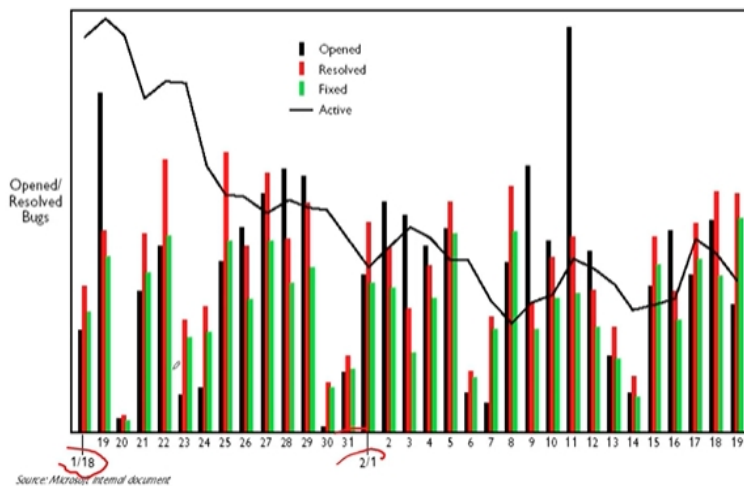
Quindi le metriche sono usate in modo limitat, e lo si capisce dal seguente grafico: sulle ascisse le date, le barre rappresentano i dati raccolti ogni giorno relativamente a tre elementi dal punto di vista della qualità del software: **barra nera** == **opened bugs** (*problemi riscontrati durante la giornata nel compilare il daily build*), **barra rossa** == **Resolved Bugs** (*post debugging ho capito qual è il problema*) e **barra verde** == **Fixed Bugs** (*ho effettivamente risolto il problema aggiustando il codice*).

La barretta contorta che attraversa il grafico indicherebbe il numero atteso di difetti attivi, ossia quelli che ancora devo scovare. Ciò che si fa tipicamente è stabilire un obiettivo di qualità legato al numero di difetti attivi: se per un certo periodo di tempo

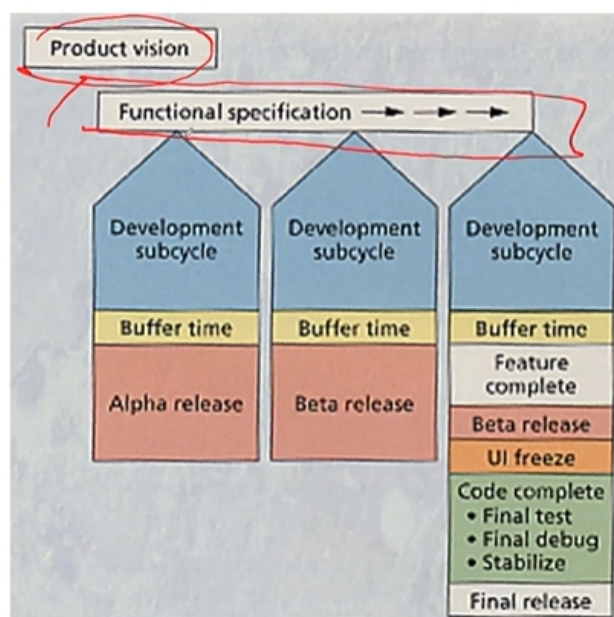
il numero di questi è inferiore alla soglia allora target di qualità raggiunto ma se non fosse il prodotto sarà comunque consegnato anche con molti difetti.

(è peggio non rispettare la data promessa al mercato che rilasciare un prodotto molto difettoso a livello di immagine dell'azienda, al prodotto difettoso si ovvia con aggiornamenti futuri)

Esempio di metriche collezionate



Modello del ciclo di sviluppo synch- and- stabilize



Nel grafico vediamo come si parta dalla vision per poi avere una specifica che va avanti durante l'intero sviluppo a differenza del waterfall. In questo caso vediamo come il progetto sia diviso in tre cicli di sviluppo, la fine di ognuno delle quali rappresenta una **milestone**. In ognuno di questi cicli si procede chiaramente con la fase di development, poi c'è un **buffer time** (nella fase iniziale si definisce lo scheduling di ciò che si farà e si usano tecniche di stima di tempi, costi.. per capire quando l'azienda sarà in grado di rilasciare il prodotto. Nonostante queste stime siano abbastanza precise, durante lo sviluppo spesso le cose non vanno come

pianificato e quindi a ogni milestone si aggiunge un buffer time per cercare di risolvere eventuali ritardi). Si ricorda che il primo sottoprogetto deve contenere le funzionalità critiche affinché se qualcosa non andasse non si è andati troppo avanti. Dopo la prima milestone esce l'**alpha release** per il testing interno, nel secondo milestone per cui si arriva già a una parte consistente di prodotto si rilascia la versione Beta e infine dopo la terza milestone si arriva a **Feature Complete** per cui non si potranno aggiungere funzionalità, poi ancora rilascio versione beta e poi cosa tipica di Microsoft la **UI freeze** (congelamento della user interface. La prima cosa su cui si lavora infatti è l'interfaccia utente, che viene verificata in laboratori di usabilità molto avanzati presenti in sede microsoft e che una volta superati i controlli "congela" ossia non potrà più essere toccata fino al rilascio). Si arriva poi al rilascio definitivo.

Confronto tra modelli *synch-and-stabilize* e *waterfall*

Synch-and-Stabilize	Sequential Development
Product development and testing done in parallel	Separate phases done in sequence
Vision statement and evolving specification	Complete "frozen" specification and detailed design before building the product
Features prioritized and built in 3 or 4 milestone subprojects	Trying to build all pieces of a product simultaneously
Frequent synchronizations (daily builds) and intermediate stabilizations (milestones)	One late and large integration and system test phase at the project's end
"Fixed" release and ship dates and multiple release cycles	Aiming for feature and product "perfection" in each project cycle
Customer feedback continuous in the development process	Feedback primarily after development as inputs for future projects
Product and process design so large teams work like small teams	Working primarily as a large group of individuals in a separate functional department

Tra le differenze molte simili a quelle viste nel confronto waterfall e incrementale, si hanno in più giusto che vi sono le milestone, che le sincronizzazioni avvengono per daily builds, che ciò che conta non è più tanto creare un progetto con meno difetti possibili come si punta a fare nei software a contratto ma si punta a mantenere pulita l'immagine dell'azienda restando nei tempi.

NB. quanto visto riguarda Microsoft anni '90 da parte di due ricercatori.

Un'ulteriore esperienza è stata fatta da Cusumano è stato con Yoffie e non Selby in un'azienda agli antipodi di Microsoft, che realizzava freeware: **Netscape**. Netscape all'epoca era la principale competitor di Microsoft per la realizzazione di Browser (già all'epoca Microsoft voleva monopolizzare il browsing con Internet Explorer preinstallato in Windows).

In Netscape già la dimensione dello staff cambiava: ogni tre sviluppatori un tester mentre in Microsoft uno a uno. Comunque in termini di produttività comparabile a Microsoft per prodotti simili.

Riguardo il processo software vi era invece scarso effort di pianificazione (eccetto che sui prodotti server che erano la fonte di guadagno dell'azienda. Infatti il profitto di Netscape derivava dai software per server web ossia quel software che risponde quando digitiamo il link e ci collega al server corrispettivo. La realizzazione di browser gratuiti era pubblicità in funzione di questo).

Inoltre il **processo era caratterizzato da documentazione incompleta** (attività considerate marginali evitate), **scarso controllo sullo stato di avanzamento** del progetto (lasciato all'esperienza e influenza dei PM), **scarso controllo su attività di ispezione del codice** (causa meno tester) e **pochi dati storici per il supporto alle decisioni**.

È stato proprio il fatto di non basarsi su dati reali come supporto alle decisioni e fidarsi dell'esperienza dei PM che ha portato al fallimento dell'azienda

Table 2. Allocation of staff in Netscape's client and server development divisions, mid-1998.

	Client products	Server products	Total
Software engineering	110	200	310
Testing (QA)	50	80	130
Product management	50	42	92
Subtotal	210	322	533
Other*	30	98	128
Total	240	420	660

Solo un terzo degli sviluppatori software su prodotti browser, il resto sui server fonte effettiva di profitto per l'azienda. 660 persone in totale di staff, nemmeno confrontabile con Microsoft, tuttavia competitor serio sul fronte browser.

Il processo di sviluppo era simile in Netscape a un synchronize-stabilize, chiaramente con effort ridotto.

Ciò che era il vision statement in microsoft qui diventa una vision generata da **Advanced Planning Meeting APM** ossia riunioni in cui si discuteva di ciò che si sarebbe potuto realizzare in base alle opportunità di mercato. In queste riunioni erano coinvolti gli **esperti di marketing, sviluppatori ed executives** (esperti imprenditoriali proprietari dell'azienda).

Da qui la **vision**, la **pianificazione delle attività di sviluppo**, si crea la **specificazione funzionale** con gli ingegneri e talvolta col supporto dei product manager (marketing) e si **pianificano le varie attività allocando budget, risorse umane e denaro** discutendone anche con gli executives.

Dopodiché lo sviluppo viene monitorato con meeting periodici (**First Executive Review**) in cui gli sviluppatori si incontrano con gli executive per informarli dello stato di avanzamento del lavoro.

Poi finita la fase di sviluppo si passa al rilascio dell'alpha e poi due beta che in questo caso venivano rilasciate a tutti permettendo feedback più ampio.

Poi si completava il codice e dopo la **stabilizzazione** e debugging rilascio RTM (release to manufacturing, copia "gold" sarebbe la versione definitiva).

Ciò che comportò il fallimento di Netscape fu proprio l'intervento degli executive: all'epoca Microsoft aveva aggiunto al Browser **active channel**, una funzionalità che permetteva l'aggiornamento automatico della pagina senza che l'utente dovesse farlo manualmente molto utile in diversi contesti.

Si fece una interim executive review verso la fine di un progetto legato all'aggiornamento del browser, i tecnici ovviamente suggerirono di implementare lo stesso immediatamente in quanto fase troppo avanzata ma gli executive si sono opposti e la versione del browser era sostanzialmente inusabile. Da qui danno così grande che l'azienda è fallita 😞.

Ultimi metodi che vediamo in questa parte sono i **Metodi Agili**.

All'inizio dei 2000 si osservò una reazione contro la formalizzazione dello sviluppo software (in particolare Waterfall). L'imporre dei modelli di sviluppo, standard da utilizzare e documentazione da produrre infatti non permette di sfruttare al massimo la creatività degli sviluppatori.

Si voleva rendere lo sviluppo del software meno pesante e più agile.

In realtà il modello Agile estende ulteriormente le caratteristiche dell'approccio incrementale puntando su una comunicazione molto intensa tra sviluppatori e cliente/utente, insistendo su fast feedback.

Quanto caratterizza i metodi agili è racchiuso nell' **Agile Manifesto** che ne definisce sia i valori che i principi. Tra i **valori**: *ciò che conta sono gli individui e le interazioni piuttosto che i processi e i tools, si punta a produrre software funzionante piuttosto che documentazione, conta la comunicazione con il cliente piuttosto che negoziare ai fini di un contratto e conta avere team che rispondono ai cambiamenti piuttosto che seguire un rigido piano di lavoro.*

Se ci pensiamo questa sembra un'estremizzazione dell'approccio Build & Fix, ma ciò che cambia e che vedremo è che **comunque esiste una sistematica organizzazione**. In aggiunta di questi valori contiene dodici principi guida ulteriori. Valori e principi definiscono i concetti di base dell' Agile Development.

Uno dei metodi agili più in auge è **Scrum**. Il concetto alla base è stato introdotto già a metà anni 80 non per il software ma per sviluppo di prodotti basati più su knowledge-management (in cui la conoscenza condivisa, il pensiero critico e la collaborazione sono essenziali per il risultato) che sul processo (dove si punta su ripetibilità e controllo, non sull'adattamento creativo).

Tre ruoli principali: **Scrum Master**, **Development Team** (3-9 persone) e il **Product Owner**.

Quattro eventi che riguardano uno **Sprint**: **Sprint Planning**, **Development Work**, **Sprint Review**, **Sprint Retrospective**.

Artefatti prodotti nella parte bassa della figura: **Product Backlog**, **Sprint Backlog**, **Definition of Done** e **Incremento**.

Lo **Scrum Master** *assicura che la metodologia sia compresa bene e correttamente implementata dal team di sviluppo e dal product owner. Egli non partecipa direttamente alle attività di sviluppo ma garantisce il rispetto delle regole scrum interagendo con i team.*

Il **Product Owner** *è la figura responsabile della definizione e della prioritizzazione dei requisiti da implementare. Gestisce il **Product Backlog**, che è l'elenco ordinato e continuamente aggiornato di tutte le attività e requisiti che costituiscono il lavoro da svolgere per lo sviluppo del prodotto.*

Il **Development team** *si occupa di sviluppo, testing, attività tecniche.*

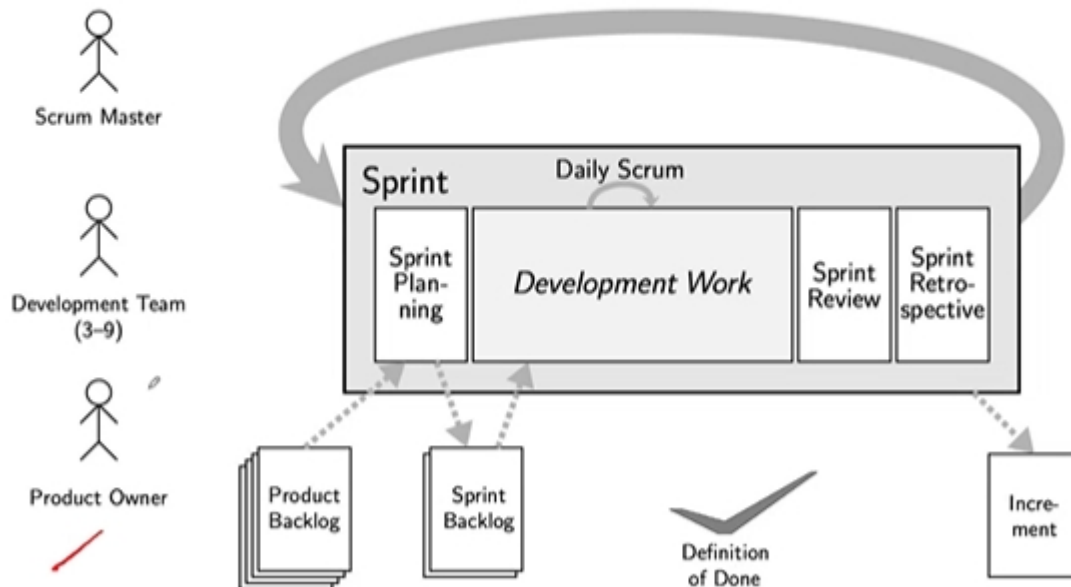
Alla base di questa metodologia c'è lo **Sprint**. *Uno sprint corrisponde all'idea di build, è quanto viene fatto per consegnare un incremento successivo del software e tipicamente richiede 2/4 settimane (quindi build molto piccoli, questo per garantire feedback continuo).*

Si inizia con lo **Sprint Planning** per cui vengono presi i requisiti nel **Product Backlog** e trasferiti nello **Sprint Backlog** (che conterrà quanto si dovrà fare per quello specifico sprint), si farà poi lavorare il team di sviluppo a questo Sprint tramite meeting quotidiani (***daily Scrum meeting***) per cercare di organizzarsi e capire se ci sono problemi.

*Alla fine dello Sprint l'incremento è presentato al cliente in una **Sprint Review**.*

*Infine vi è la **Sprint Retrospective** per confrontare quanto fatto con gli obiettivi che si avevano inizialmente di modo da pianificare il prossimo Sprint includendoci gli obiettivi non raggiunti (piuttosto che consegnare più tardi lo Sprint posticipo le attività mancanti nel successivo, concetto di **timeboxing**).*

Scrum



Scrum ha poi bisogno di un **Definition of Done**, ossia un modo di capire se quanto fatto va bene o meno. Negli approcci convenzionali vi sono metodi di convalida che permettono di capire se quanto fatto è corretto o meno, qui invece sempre secondo l'idea di self-organizing è lo stesso team di sviluppo a definire per se stesso cosa significa aver raggiunto o meno l'obiettivo.

In definitiva con Scrum si lavora in modo veloce ed agile anche se una minima forma di processo e organizzazione del lavoro viene ovviamente imposta al team di sviluppo.

Lez 8 (26/10)

Un aspetto importante che vedremo anche nella parte di *Definizione dei Requisiti* è il concetto di **User Story**. Si tratta di una pratica comune utilizzata nello sviluppo Agile, spesso in combinazione con Scrum, per far interagire in modo più efficace gli utenti nello sviluppo software.

Infatti l'obiettivo principale dell'ingegnere del software è aiutare gli sviluppatori a costruire software di qualità e qualità intesa come capacità di soddisfare le esigenze degli utenti. L'utente è colui che userà il prodotto e per questo è l'elemento principale su cui focalizzarsi.

Invece di chiedere all'utente come vorrebbe il prodotto, l'idea è di definire questi requisiti ad alto livello iniziali con il concetto di User Story.

Lo user story è un formato per definire il requisito utente come una "storia", che deve essere breve (tipicamente una sola frase) e descrive il punto di vista dell'utente. Si usa il formato standard:

- As a *<role>*, I want *<goal>* so that *<benefit>*
 - Example: “As a process engineer, I want to see the dependencies between different process steps so that I can easily verify and validate them”

Chiaramente non tutti i requisiti possono essere descritti in questo modo, quelli più complessi descritti da una user story più grande suddivisibile in pezzi più piccoli detti Epics.

Queste user story sono proprio ciò che popola in Scrum il **product backlog** e lo **sprint backlog**.

L'obiettivo di tutti questi modelli è stato quello di organizzare le attività di sviluppo, di modo da evitare l'approccio **Build & Fix** che tanti danni ha fatto nel '39, senza essere vincolati dalla rigida sequenza **waterfall & company**.

L'uso dei metodi Agili può sembrare tornare al Build & Fix ma non è così, infatti anche questi metodi nonostante siano più “leggeri” rispetto a un waterfall sono comunque organizzati in modo sistematico.

Come abbiamo visto purtroppo non è possibile avere per un software delle “etichette” che ne certifichino la qualità come si fa con i prodotti normali (es. DOCG), ciò rappresenta un problema perché se esistessero sarebbe più facile, si potrebbero seguire passaggi standard nello sviluppo per ottenere delle certificazioni e qualità. Per ovviare il problema l'idea è offrire certificazioni non per il software, ma per le organizzazioni che lo sviluppano. Il modello standard in questo senso è il CMM.

CMM == Capability Maturity Model rappresenta un modello introdotto nel '93 dal SEI per determinare il livello di maturità del processo software di un'organizzazione (ossia una misura dell'efficacia globale nell'uso di tecniche di ISW).

Con il tempo sono nati anche altri modelli più specifici per definire certificati di questo tipo, ed attualmente si certifica utilizzando il **CMMI == CMM Integrated** che integra tutti queste altre certificazioni.

Parlando del **CMM**, il modello è basato su un questionario ed uno schema valutativo a 5 livelli additivo (per cui se mi certifico ad un certo livello sono automaticamente certificato anche per i livelli inferiori).

Lvl 1 Initial è iniziale certifica di base ogni possibile organizzazione. “**Success depends on heroes**” nel senso che il successo dipende dalle persone specializzate, non esiste un vero e proprio processo organizzativo.

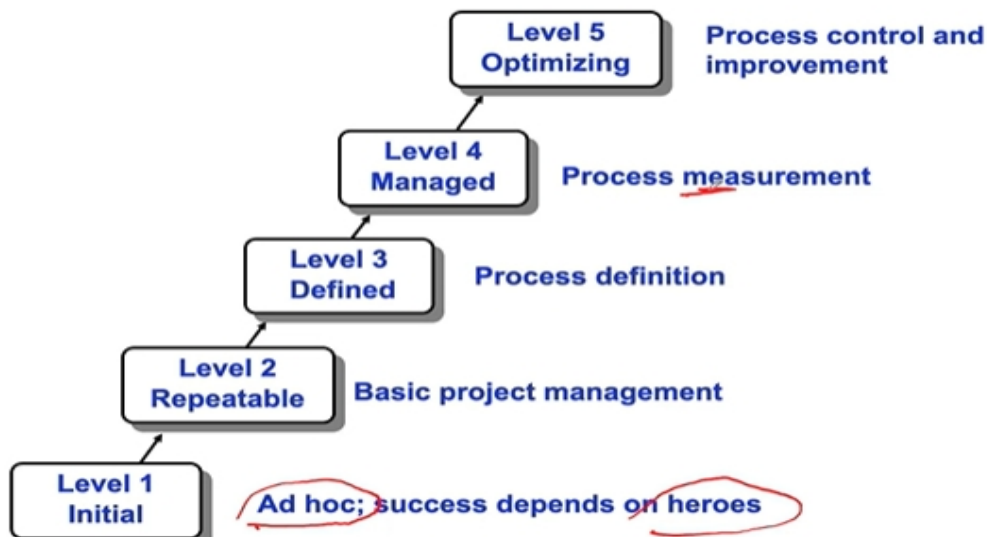
Lvl 2 Repeatable: si utilizzano tecniche di base di project management che permettono la ripetibilità (se ho fatto successo in passato con un certo approccio posso replicare quell'approccio, e ciò posso farlo solo se ho un minimo di organizzazione come pianificare lo sviluppo, monitorare..)

Lvl 3 Defined: l'organizzazione si è dotata di un Processo documentato, standardizzato e integrato per lo sviluppo software

Lvl 4 Managed: a questo livello si hanno delle *tecniche di misurazione del processo* per capire come procede quello specifico processo

Lvl 5 Optimizing: oltre a poter misurare il processo standardizzato, posso anche migliorarlo.

I 5 livelli del CMM



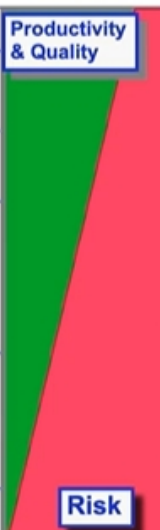
Per certificarsi un'organizzazione accede alla documentazione del SEI per capire cosa fare per arrivare a un certo livello e una volta che si sente pronta richiede di essere valutata. Infatti il SEI ha definito in modo dettagliato tutta la documentazione necessaria di modo che le aziende che vogliono certificarsi sappiano cosa devono fare per ottenere un certo livello.

Infatti il SEI associa ad ogni livello alcune **KPA Key Process Area** (in totale 18) che **descrivono le funzioni che devono essere presenti per garantire l'appartenenza a quel livello**. Ogni KPA è descritta in termini di obiettivi, responsabilità, capacità e risorse necessarie, attività da realizzare, metodi di monitoring della realizzazione, metodi di verifica della realizzazione.

L'importanza di certificarsi sta nel fatto che in vari domini applicativi viene richiesta come vincolo proprio la certificazione CMMI.

Più facile certificarsi per piccole organizzazioni in quanto toccare l'organigramma di grandi organizzazioni per adattarlo alle KPA è molto complesso.

CMM KPAs

			Result
Level	Characteristic	Key Process Areas	Productivity & Quality
Optimizing (5)	Continuous process capability improvement	Process change management Technology change management Defect prevention	
Managed (4)	Product quality planning; tracking of measured software process	Software quality management Quantitative process management	
Defined (3)	Software process defined and institutionalized to provide product quality control	Peer reviews Intergroup coordination Software product engineering Integrated software management Training program Organization process definition Organization process focus	
Repeatable (2)	Management oversight and tracking project; stable planning and product baselines	Software configuration management Software quality assurance Software subcontract management Software project tracking & oversight Software project planning Requirements management	
Initial (1)	Ad hoc (success depends on heroes)	"People"	Risk

NB essendo un modello additivo per certificarmi a livello 3 devo soddisfare anche le KPA a livello 2.

A lvl 2 devo dimostrare di eseguire correttamente 6 delle 18 KPA definite: tra queste software configuration management (capacità di gestire la configurazione dei prodotti come controllo delle versioni), software quality assurance (attività di verifica e convalida), software subcontract management (se prendo un contratto di grandi dimensioni devo poter appaltare parti del progetto a terze parti garantendo al contempo la qualità di quanto fatto da loro) etc...

A lvl 3 Peer Reviews (tecniche formali di verifica), Training Program (corretta formazione del personale) etc...

I livelli 4 e 5 maturità elevata, tipicamente le aziende si fermano a livello 3.

A lvl 5 addirittura Defect Prevention: si è in grado di utilizzare approcci formali per prevenire l'introduzione di difetti nel software.

A destra la colonna ci dice che più basso stai col livello più grande è la percentuale di rischio. Man mano che si alza il livello aumenta la produttività e la qualità.

18 KPA da verificare per arrivare al lvl 5.

Statistiche a Febbraio 2000

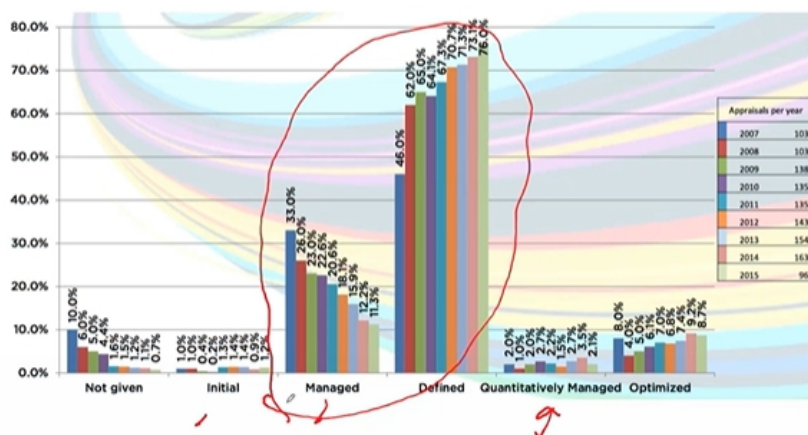
La lista delle organizzazioni a livello 4 e 5 (*maturità elevata*) include:

- 71 organizzazioni negli USA
 - 44 organizzazioni a Livello 4 (tra cui Oracle, NCR, Siemens Info Systems, IBM Global Services)
 - 27 organizzazioni a Livello 5 (tra cui Motorola, Lockheed-Martin, Boeing, Honeywell)
- 25 organizzazioni al di fuori degli USA
 - 1 organizzazione a Livello 4 in Australia
 - 14 organizzazioni a Livello 4 in India
 - 10 organizzazioni a Livello 5 in India

Le statistiche suggeriscono come lvl 5 per aziende che necessitano software sicuri per attività critiche (vedi Boeing). Chiaramente si certifica il reparto software dell'azienda e non l'azienda nella sua interezza.

24 su 25 organizzazioni in India: si tratta soprattutto di porzioni periferiche di organizzazioni statunitensi più grandi che sfruttano i poveri indiani (non sto scherzando).

Quindici anni dopo la certificazione è diventata sempre più richiesta e quindi migliaia di certificazioni.



(2015) La maggior parte delle organizzazioni sono certificate a lvl 3 perché per buona parte dei domini di mercato il vincolo è quello.