INTRODUZIONE ALL'INGEGNERIA DEL SOFTWARE

Programma = Implementazione di un algoritmo: acquisisce dati di input, elabora, presenta dati di output.
Sistema software = Insieme di componenti software che funzionano in modo coordinato allo scopo di informatizzare una certa attività ("**soluzione informatica").** Oltre ad essere un insieme di componenti software il sistema software è formato da file di configurazione + file di dati + file di installazione + librerie + documentazione di sistema + documentazione utente (+ sito web). Il sito web è facoltativo.

Sviluppo di un programma

Le funzionalità o i problemi che il programma deve gestire o risolvere sono relativamente semplici. In genere sono coinvolte alcune persone oppure solo una persona ed inoltre il tempo di sviluppo è piuttosto limitato (un giorno o alcuni giorni).

La fase della realizzazione si divide in:
1)Comprensione del problema che il programma deve risolvere,
2)Definizione dei possibili dati di input e output,
3)Definizione delle strutture dati necessarie (variabili, vettori, ...),
4)Definizione dell'algoritmo che risolve il problema,
5)Scrittura del codice corrispondente,
6)Testing.

Sviluppo di un sistema software

Le funzionalità del sistema sono più complesse e articolate rispetto a quella di un programma.

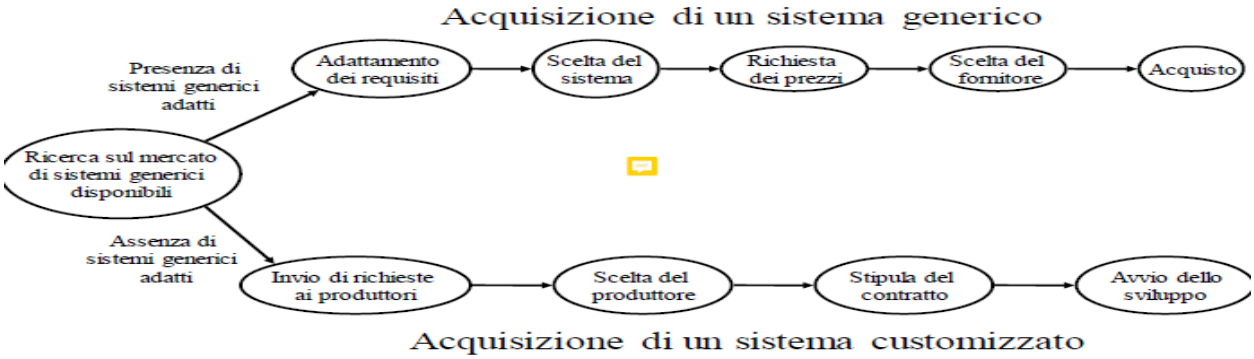
La realizzazione di un sistema software richiede l'impiego di un **gruppo di lavoro** in cui ogni persona ha un proprio ruolo. Il gruppo può contenere sottogruppi con funzioni specifiche e che interagiscono tra di loro poiché le attività dei vari sottogruppi devono essere coordinate. La fase di sviluppo richiede molto **tempo** e sono necessarie varie fasi di sviluppo.
Da notare che le attività devono essere pianificate (risorse umane e tecniche necessarie, dipendenze tra attività, possibili rischi, ...) e bisogna stimare i tempi di consegna.

Esistono due tipi di sistemi software generici e customizzati.

1)sistema generico: Tali sistemi sono messi sul mercato e venduti ad un qualsiasi cliente. I requisiti sono definiti in base alle tendenze di mercato. L'obiettivo è vendere il maggior numero di copie.
2)sistema customizzato: Tali sistemi sono richiesti da uno specifico cliente detto **committente.** I requisiti sono definiti in base alle richieste del committente e l'obiettivo è soddisfare il committente (viene fatto un sistema ''su misura''). Il costo di sviluppo è interamente a carico del committente (più costoso).

Esistono anche dei sistemi che possono essere ibridi ovvero nascono come generici e poi vengono customizzati poi per un certo cliente.

ACQUISIZIONE DEL SISTEMA DA PARTE DI UN CLIENTE/COMMITENTE PER SISTEMA CUSTOMIZZATO



NASCITA DELL'INGEGNERIA DEL SOFTWARE

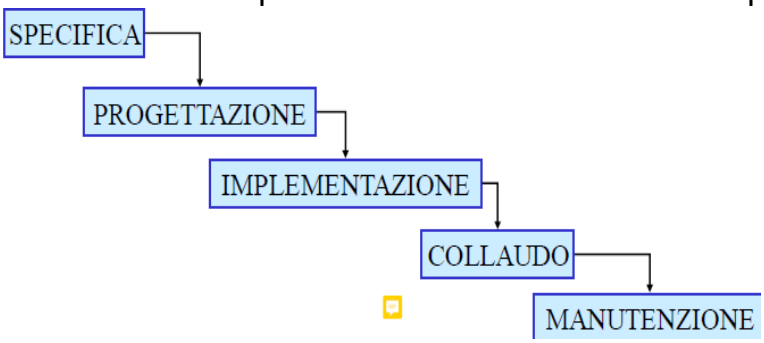
L'Ingegneria del Software nasce intorno al 1968 come risposta alla crisi del software. Nella crisi del software le tecniche usate per i programmi non erano adatte ai sistemi software e questo generava diversi problemi. Per esempio, il sistema non forniva i servizi richiesti dal committente oppure l'uso del sistema era difficile da comprendere per l'utente oppure le modifiche del sistema per correggere eventuali errori o migliorare dei servizi erano difficili da realizzare.

Ingegneria del Software: applicazione del processo dell'Ingegneria alla produzione di sistemi software.
L'Ingegneria del Software dovrebbe limitare la presenza di difetti alla consegna del sistema software; fare in modo che il sistema soddisfi il committente cioè il sistema deve fornire i servizi richiesti. Infine, il sistema deve avere una certa qualità.
DEFINIZIONE ING SW: È il processo attraverso il quale si va a creare un sistema software. Tale processo ha il compito di limitare la presenza di difetti alla consegna del sistema sw, deve fare in modo che il sistema soddisfi tutti i requisiti richiesti dal cliente ed infine il sistema deve avere una certa qualità. Le 5 fasi che compongono tale processo sono indicate qui sotto.

LE 5 FASI DEL PROCESSO DI SVILUPPO IN INGEGNERIA DEL SOFTWARE

1)SPECIFICA: In tale fase si definiscono i requisiti che possono essere:
-I requisiti funzionali che definiscono funzioni e servizi che il sistema deve offrire (per esempio legati al cliente possono essere il login, l'acquisto di un prodotto, il pagamento di un prodotto).
-I requisiti non funzionali sono gli attributi di qualità, affidabilità, strumenti e linguaggi.
2)PROGETTAZIONE in tale fase si definisce l'architettura del sistema, il comportamento dei componenti coinvolti, le strutture dati, gli algoritmi, la struttura del codice e l'interfaccia utente del sistema.
3)IMPLEMENTAZIONE in tale fase si vanno a realizzare tutti i componenti progettati del sistema ed integrarli. Questa fase prevede la scrittura del codice e l'integrazione dei moduli.
4)COLLAUDO tale fase si suddivide in:
-Fase di verifica in cui si fanno dei test sul sistema e si controllano che non ci siano difetti di funzionamento.
-fase di validazione in cui si controlla che tutti i requisiti siano stati realizzati.
5)MANUTENZIONE: tale fase si suddivide in tre fasi che avvengono tutte e tre dopo la consegna del sistema:
-fase correttiva in cui si correggono difetti emersi dopo la consegna (non scoperti nel collaudo)
-fase migliorativa soddisfare nuovi requisiti o migliorare gli esistenti
-fase adattiva adattare il sistema a cambiamenti di hardware o sistema operativo

Modello di processo: è un tipo di organizzazione delle fasi del processo software. Inizialmente assumiamo che valga il **modello a cascata**: in cui le fasi sono distinte e sequenziali (a cascata) e prima di passare alla fase successiva, la fase corrente deve essere stata completata. Esistono altri modelli di processo.



GESTIONE DEL PROCESSO

L'Ingegneria del software si occupa anche della **gestione del progetto** che si svolge in parallelo al processo software.

Le principali **attività di gestione** sono:

–Assegnare risorse umane, tecnologiche, finanziarie ad ogni attività

–Stima del tempo necessario per ogni attività

–Stima dei costi di ogni attività

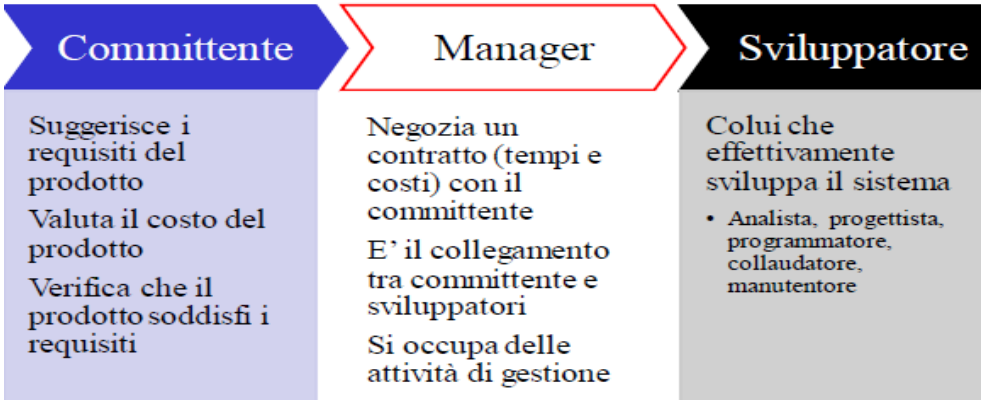
–Stima dei rischi (situazioni avverse che si possono presentare)

Gli **Obiettivi** della gestione sono che:

–Il sistema deve essere consegnato al committente nei tempi previsti

–Il sistema deve avere un costo finale non superiore a quello previsto

Personaggi chiave



Qual è la differenza tra informatica e ingegneria del software?

L'informatica si occupa dei principi teorici alla base del software (programmazione, sistemi operativi, basi di dati, reti, …) mentre l'ingegneria del software si occupa del processo di sviluppo (specifica, progettazione, implementazione, collaudo, manutenzione) e delle attività di gestione.

L'informatica non è sufficiente per realizzare un sistema software; servono anche:

1)Ingegneria del software.

2)La conoscenza del dominio applicativo ovvero si deve avere una conoscenza di quello che tratta il sistema, ad esempio, se il sistema deve gestire una banca bisogna conoscere come fare un bonifico o come funzionano le azioni bancarie ecc.

3)Capacità creativa per trovare soluzioni ad-hoc.

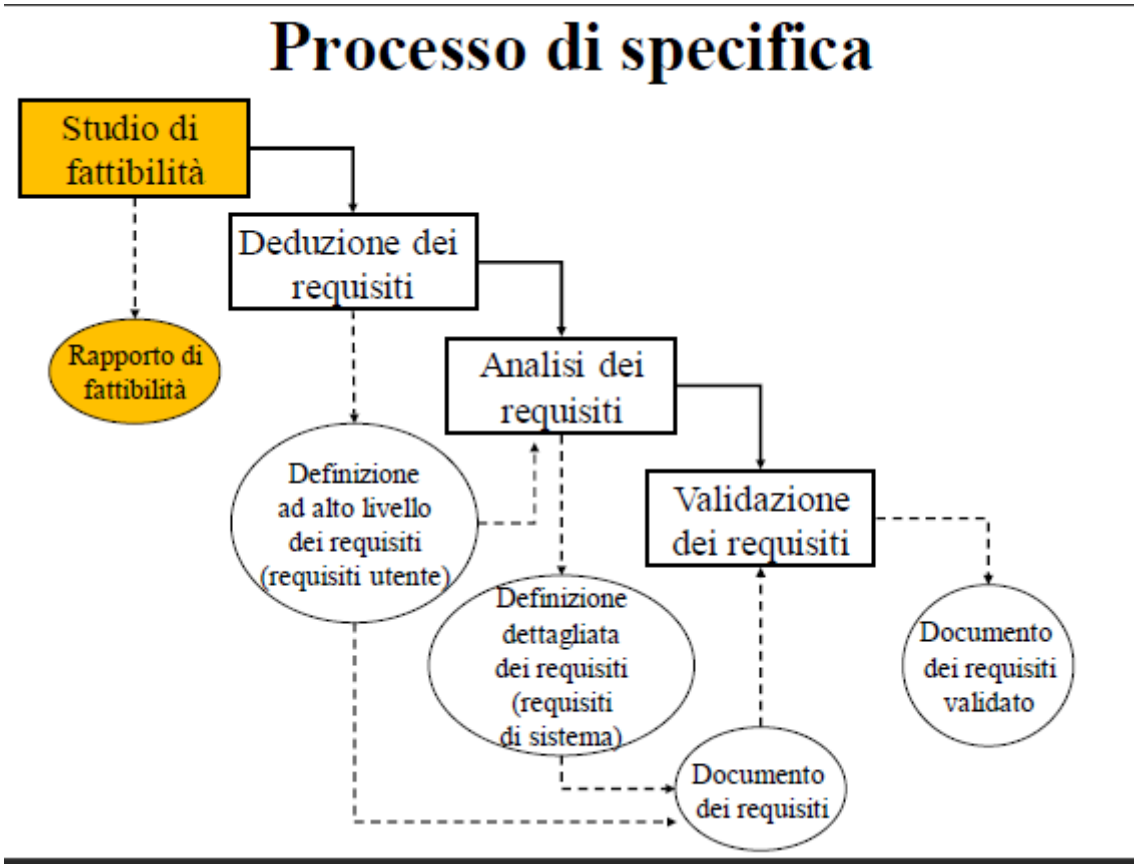
STRUMENTI CASE: sono tutti quei tool che mi supportano nelle varie fasi del ciclo di vita del software. Di seguito degli esempi di tool che mi supportano nelle varie fasi.
1)Specifica: editor dei requisiti (testo e UML), ...

2)Progettazione: editor dei diagrammi di progettazione (UML), ...

3)Implementazione: IDE, strumenti per condivisione dei file, ...

4)Collaudo: strumenti per ispezione e testing automatico del codice, ...

5)Manutenzione: strumenti per la gestione delle versioni del codice e della documentazione.



FASE DI SPECIFICA
La fase di specifica stabilisce "cosa" il sistema deve fare, non "come" funziona al suo interno. Quindi stabilisce i requisiti che possono essere funzionali o non funzionali servono per una proposta di contratto e per le fasi successive del processo software.

1)Requisiti funzionali: servizi (funzioni) che il cliente/committente richiede dal sistema.

2)Requisiti non funzionali: non descrivono i servizi (funzioni) del sistema, ma descrivono la qualità generale e i vincoli di sviluppo.

Il grado di dettaglio di un requisito può variare tra una descrizione astratta del comportamento del sistema, ed una descrizione formale-matematica.

Requisiti funzionali
Per ogni servizio, si descrive
1)Cosa accade nell'interazione tra utente e sistema,

2)Cosa accade in seguito ad un certo input o stimolo,

3)Cosa accade in particolari situazioni, ad esempio in caso di eccezioni.
Non si descrive **come** funziona **internamente** il sistema poiché i componenti, la loro architettura, e il loro comportamento non si conoscono ancora perché essi sono oggetto della successiva fase di **progettazione.**

Requisiti non funzionali
I requisiti non funzionali si dividono in tre sottogruppi di requisiti:
1)Requisiti di prodotto: sono gli attributi che definiscono la **qualità** del sistema (Efficienza, Affidabilità, Usabilità, Mantenibilità, Portabilità, Recoverability). Sono detti anche **proprietà emergenti e complessive.**

Da qui fuori escono due proprietà:

- **Proprietà complessiva:** riguarda il sistema nel suo complesso, non riguarda un particolare componente o una particolare funzione.
- **Proprietà emergente**: proprietà che "emerge" dal funzionamento del sistema, dopo che è stato implementato.

Vediamo una definizione degli attributi che definiscono la qualità del sistema:
Usabilità: definisce il grado di facilità con cui l'utente riesce a comprendere l'uso del software. Il sistema deve avere un'**interfaccia utente** intuitiva e curata, inoltre l'uso del sistema deve essere ben documentato.
Efficienza: è il livello di prestazioni del sistema. Si misura in vari modi; per esempio, in tempo di risposta, il numero medio di richieste soddisfatte per unità di tempo, la quantità di risorse consumate e l'occupazione di memoria in termini di KB, MB o GB.
Affidabilità: è il grado di fiducia con cui si ritiene che il sistema svolga in modo corretto la propria funzione. Ci sono vari tipi (o misure) di affidabilità:

Reliability: che è la capacità di fornire i servizi in modo continuativo per una certa durata di tempo.

Availability: è la capacità di fornire i servizi nel momento richiesto.

Safety: è la capacità di operare senza causare danni materiali alle persone, all'ambiente, alle infrastrutture.

Security: è la capacità di proteggersi da intrusioni e attacchi.

Mantenibilità: è il grado di facilità di manutenzione. Deve essere possibile l'evoluzione del software per soddisfare i requisiti nel tempo.
Portabilità: è la capacità di migrazione da un ambiente ad un altro.
Recoverability: è la capacità di ripristinare lo stato e i dati del sistema dopo che si è verificato un fallimento (crash).
2)Requisiti organizzativi: sono caratteristiche riguardanti le fasi del processo software o la gestione del progetto. Si suddividono in:

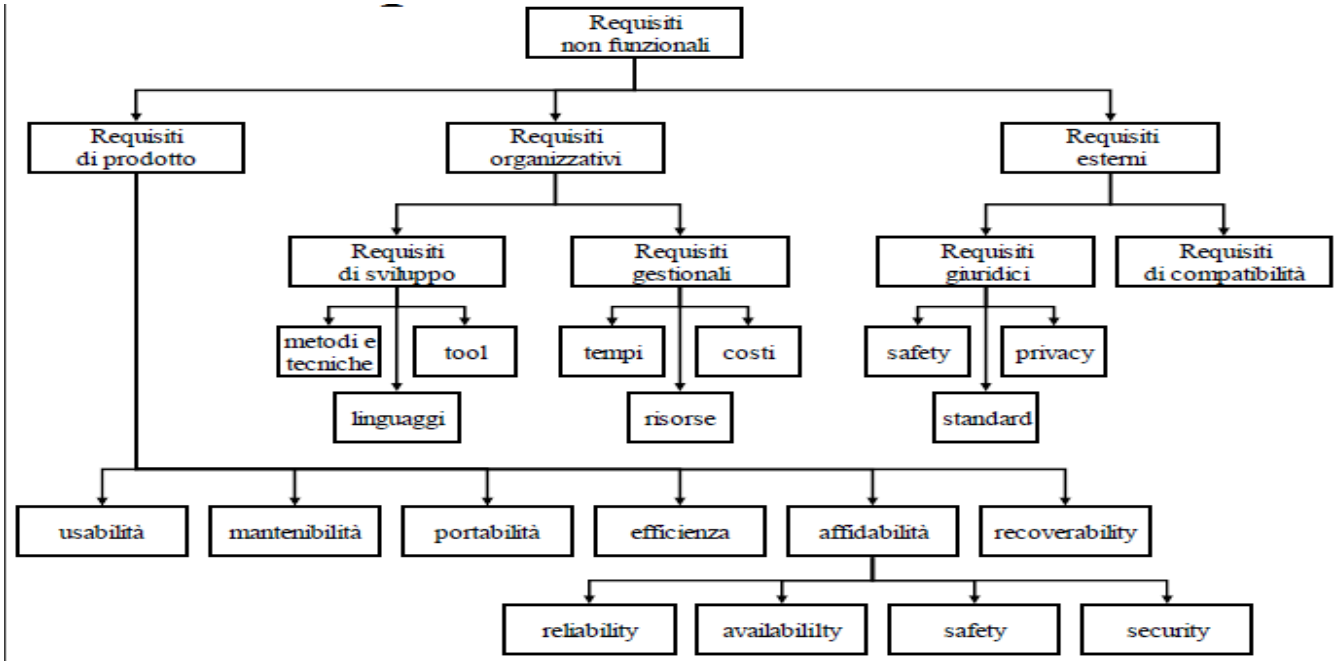
Requisiti di sviluppo: metodi e tecniche di sviluppo, linguaggi di programmazione, strumenti CASE, …, da adottare durante il processo sw.

Requisiti gestionali: tempi, risorse, costi

3)Requisiti esterni: derivano da fattori esterni al sistema e al processo software. Si suddividono in:

Requisiti di compatibilità con altri sistemi

Requisiti giuridici (rispetto di leggi sulla privacy o sulla sicurezza, rispetto di standard)



Sistemi critici

Un sistema è critico quando il suo non corretto funzionamento può provocare conseguenze "disastrose". I tipi di conseguenze critiche sono:

1)Safety critical system: un fallimento (crash) del sistema può provocare danni alle persone o all'ambiente come, per esempio, il sistema di controllo del traffico aereo.

2)Business critical system: un fallimento (crash) del sistema può determinare perdite economiche come, per esempio, il sistema per la gestione delle operazioni bancarie.

Costo dell'Affidabilità

L'Affidabilità è la proprietà più importante e il suo costo cresce in modo esponenziale rispetto al grado di Affidabilità richiesto.

Le tecniche di sviluppo più costose per soddisfare l'Affidabilità, oltre che i requisiti funzionali sono:

1)Tecniche di tolleranza ai guasti (fault-tolerance)

2)Tecniche per proteggersi da attacchi

Il maggiore costo è giustificato dalla riduzione del rischio di danni dovuti al funzionamento non corretto del sistema critico.

Per concludere diciamo che dobbiamo trovare un equilibrio tra affidabilità, funzionamento ed efficienza di un sistema.

Processo di specifica
Il processo di specifica si suddivide in quattro fasi:

1)Studio di fattibilità: è la valutazione della possibilità di sviluppare il sistema e dei suoi vantaggi per il committente
2)Deduzione dei requisiti: è la raccolta di informazioni da cui dedurre quali sono i requisiti. In questo passo si ottiene la definizione ad alto livello dei requisiti detti **requisiti utente**.
3)Analisi dei requisiti: organizzazione, negoziazione e modellazione dei requisiti. In questo passo si ottiene la definizione dettagliata dei requisiti detti **requisiti di sistema**.
4)Validazione dei requisiti: è la verifica del rispetto di alcune proprietà da parte del documento dei requisiti. A questo passo si ottiene il documento dei requisiti validato che contiene i requisiti descritti con due livelli di dettaglio: utente e di sistema.

Differenza tra requisito utente e requisito si sistema

1)Requisito utente: è un requisito descritto ad alto livello ed è:

-Il risultato della **deduzione dei requisiti**

-Viene letto da persone senza un'approfondita conoscenza tecnica come il committente, utenti finali, …

-Viene espresso in linguaggio naturale facendo uso di scenari + diagrammi molto semplici.

2)Requisito di sistema: requisito descritto dettagliatamente

-Risultato dell'**analisi dei requisiti**

-Fornisce tutti i dettagli necessari per la fase di progettazione: input, output, eccezioni, ecc

-Viene letto da persone con competenze tecniche: sviluppatori del sistema, personale tecnico presso il committente, ...

-Viene Espresso in linguaggio formale: template, UML, notazioni matematiche, data-flow

Primo passo del processo di specifica: Lo studio di fattibilità

In questo passo si decide:
1)Se è possibile costruire il sistema? Dobbiamo verificare se le risorse tecnologiche, umane, finanziarie a disposizione sono sufficienti e se i tempi e i costi richiesti sono accettabili.
2)Se il sistema è effettivamente utile al cliente o committente? ovvero dobbiamo vedere se è utile informatizzare una certa attività o esistono altri modi.

Per rispondere alle due domande qui sopra si raccolgono informazioni.
Le informazioni vengono prese dal committente, utenti finali, sviluppatori che hanno lavorato su casi simili.

Infine, dopo aver raccolto informazioni si prepara un **rapporto di fattibilità** in cui indichiamo:
1)Valutazione della possibilità di costruire il sistema
2)Valutazione dei vantaggi derivanti dalla sua introduzione presso il cliente/committente
3)Proposte di cambiamenti agli obbiettivi, ai costi, ai tempi, …

Secondo passo del processo di specifica: La deduzione dei requisiti

Attraverso la deduzione si compie una sorta di indagine per "dedurre" i requisiti. Si stabiliscono i requisiti raccogliendo informazioni da:

1)Dallo studio del dominio applicativo del sistema richiesto.
2)Dal dialogo con stakeholder.
3)Dallo studio di sistemi già realizzati e con lo stesso dominio applicativo o con un dominio simile.
4)Dallo studio dei sistemi con cui dovrà interagire quello da sviluppare.

1)Lo studio del dominio applicativo del sistema richiesto.

Il dominio applicativo è un insieme di entità reali su sui il sistema software viene applicato.

2)Dal dialogo con stakeholder.

Stakeholder (in ambito economico): è un soggetto che può influenzare il successo di un'impresa o che ha interessi nelle decisioni dell'impresa.
–azionisti, finanziatori, amministratori, dipendenti, clienti, fornitori, ecc

Stakeholder (nel processo software): persone che possono influenzare il processo o che hanno interesse nelle decisioni assunte in esso:
-sviluppatori che hanno lavorato su sistemi simili, esperti del dominio, utenti finali, cliente/committente, ecc

Per ottenere informazioni dagli stakeholder si usano:
Interviste: in cui viene richiesto di raccontare attraverso degli esempi reali come l'attività lavorativa funziona realmente, o come si immagina di interagire con il sistema.
Etnografia (''descrizione del popolo''): si osservano i potenziali utenti nello svolgimento delle loro mansioni così facendo riusciamo a capire come si svolgono le attività all'interno della realtà lavorativa.

Durante il dialogo con gli stakeholder possono insorgere dei problemi:

1)Non sanno indicare chiaramente cosa vogliono dal sistema.
2)Omettono informazioni che ritengono ovvie, ma che non lo sono per lo sviluppatore
3)Utilizzano la terminologia del dominio di cui sono esperti. Terminologia diversa da quella dello sviluppatore.
4)Lo stesso requisito può essere espresso da più stakeholder in modo diverso e in questo caso bisogna capire se le varie espressioni del requisito concordano o differiscono.
5)Stakeholder diversi potrebbero fornire requisiti in conflitto.
6)Possono trovarsi in sedi lontane.
7)Potrebbero avere poco tempo per interagire.

Inoltre, il linguaggio naturale può portare a dei problemi di:
1)Mancanza di chiarezza ovvero non si dà un'idea precisa del requisito.
2)Confusione ovvero non si distinguono bene requisiti funzionali e requisiti non funzionali.
3)Mescolanza dei requisiti.

Delle possibili soluzioni a questi problemi sono:
1)Usare un linguaggio evitando sinonimi,
2)Evitare un gergo troppo tecnico,
3)Utilizzare la formattazione del testo,
4)Descrivere i requisiti usando degli scenari,
5)Descrivere i requisiti usando dei diagrammi semplici come gli schemi a blocchi.

Terzo passo del processo di specifica: Analisi dei requisiti

Tale fase consente di passare dai requisiti utente ai requisiti di sistema che saranno poi utili nella fase di progettazione. Tale fase comprende:

1)Classificazione e organizzazione dei requisiti i requisiti correlati sono raggruppati secondo la classificazione dei requisiti.
2)Assegnazione di priorità ai requisiti: si stabilisce il grado di rilevanza di ogni requisito (alcuni requisiti sono essenziali, altri potrebbero essere opzionali)
3)Negoziazione dei requisiti:
•individuazione di conflitti tra requisiti;
•individuazione dei requisiti non realizzabili in base alle risorse disponibili
•Stakeholder e analisti stabiliscono dei compromessi sui requisiti per risolvere i conflitti o per rendere i requisiti realizzabili in base alle risorse
4)Modellazione analitica dei requisiti: produzione di modelli che rappresentano o descrivono **nel dettaglio** i requisiti (es.: template, diagrammi UML, notazioni matematiche, …)

Requisiti di sistema
Sono un'espansione dei requisiti utente.
Il linguaggio naturale non è adatto alla definizione di un requisito di sistema poiché ci può essere:

- **Ambiguità**: una parola può essere interpretata in più modi
- **Lunghezza**: la descrizione precisa e dettagliata di un requisito in linguaggio naturale diventa troppo lunga

Si usano diverse forme di rappresentazione:

- Linguaggio naturale strutturato: template
- Modelli grafici: UML
- Notazione basata su trasformazione funzionale: data-flow
- Notazione matematica

Modello data-flow
Detto anche “pipe & filter”:
Filter: è un processo di trasformazione funzionale in cui dato un input si produce un output.
Pipe: trasferisce un dato da un filter ad un altro (flusso dei dati) - l’output di un filter è l’input di un altro filter.
Il flusso e l’elaborazione dei dati possono essere sequenziali o in parallelo. Il data-flow non prevede la gestione degli errori. Un errore può determinare l’interruzione del flusso di dati.
Infine, occorre dire che l’elaborazione è di tipo **batch**: input -> elaborazione ->output

Quantificare i requisiti non funzionali
I requisiti non funzionali si possono specificare definendone delle misure quantitative:
Efficienza:
- Numero di transazioni elaborate al secondo
- Tempo di risposta
- Occupazione di memoria in termini di KB, MB o GB

Affidabilità:
- Probabilità di malfunzionamento
- Tempo medio di funzionamento corretto
- Probabilità di indisponibilità
- Tempo medio di disponibilità

Usabilità:
- Tempo di addestramento
- Numero di finestre di Help

Documento dei requisiti
Contiene il risultato della deduzione e dell’analisi e nel documento ogni requisito è descritto sia ad alto livello sia a basso livello. Questa è la dichiarazione ufficiale di ciò che si deve sviluppare
La **Struttura generale** per un documento dei requisiti:
1)Introduzione: Breve descrizione delle funzionalità del sistema
2)Glossario: definizione dei termini tecnici e delle sigle usate nel documento
3)Requisiti utente (funzionali e non funzionali)
4)Requisiti di sistema (funzionali e non funzionali)
-Identificatore, Nome, Tipo (funzionale / non funzionale), Priorità, Descrizione, Modelli (UML)

Chi legge il documento dei requisiti?

1)Committente: verifica che i requisiti corrispondano a tutte le sue esigenze (fase di validazione dei requisiti).

2)Manager di progetto: basa sul documento le attività di gestione (stima dei costi, dei tempi, dei rischi, assegnazione delle risorse).

3)Progettisti: i requisiti di sistema sono la base per fare la progettazione.

4)Collaudatori: devono controllare che il sistema implementato soddisfi i requisiti (oltre a verificare l'assenza di malfunzionamenti).

5)Manutentori: leggono il documento per capire gli scopi del sistema (non è detto che siano le stesse persone che hanno realizzato il sistema).

Validazione dei requisiti

In questa fase si verifica che il documento dei requisiti soddisfi una serie di proprietà. La validazione dei requisiti serve ad evitare la scoperta di **errori di specifica** durante le fasi successive del processo software.

–Bisogna evitare che gli errori si propaghino alle fasi successive del processo software.

–Modificare la progettazione o l'implementazione può essere molto costoso

Proprietà da verificare

1)Completezza: tutti i requisiti richiesti dal committente devono essere documentati

2)Coerenza: la specifica dei requisiti non deve contenere definizioni tra loro contraddittorie

3)Precisione: l'interpretazione di una definizione di requisito deve essere unica (non ambigua)

4)Realismo: i requisiti possono essere implementati date le risorse disponibili (tecnologiche, umane, finanziarie)

5)Tracciabilità: un requisito è tracciabile quando si riesce a

1)reperire la fonte di informazione relativa al requisito (**tracciabilità della sorgente**)

2)individuare i requisiti dipendenti (**tracciabilità dei requisiti**)

3)individuare i componenti del sistema che realizzano il requisito (**tracciabilità del progetto** – riguarda le fasi successive del processo software)

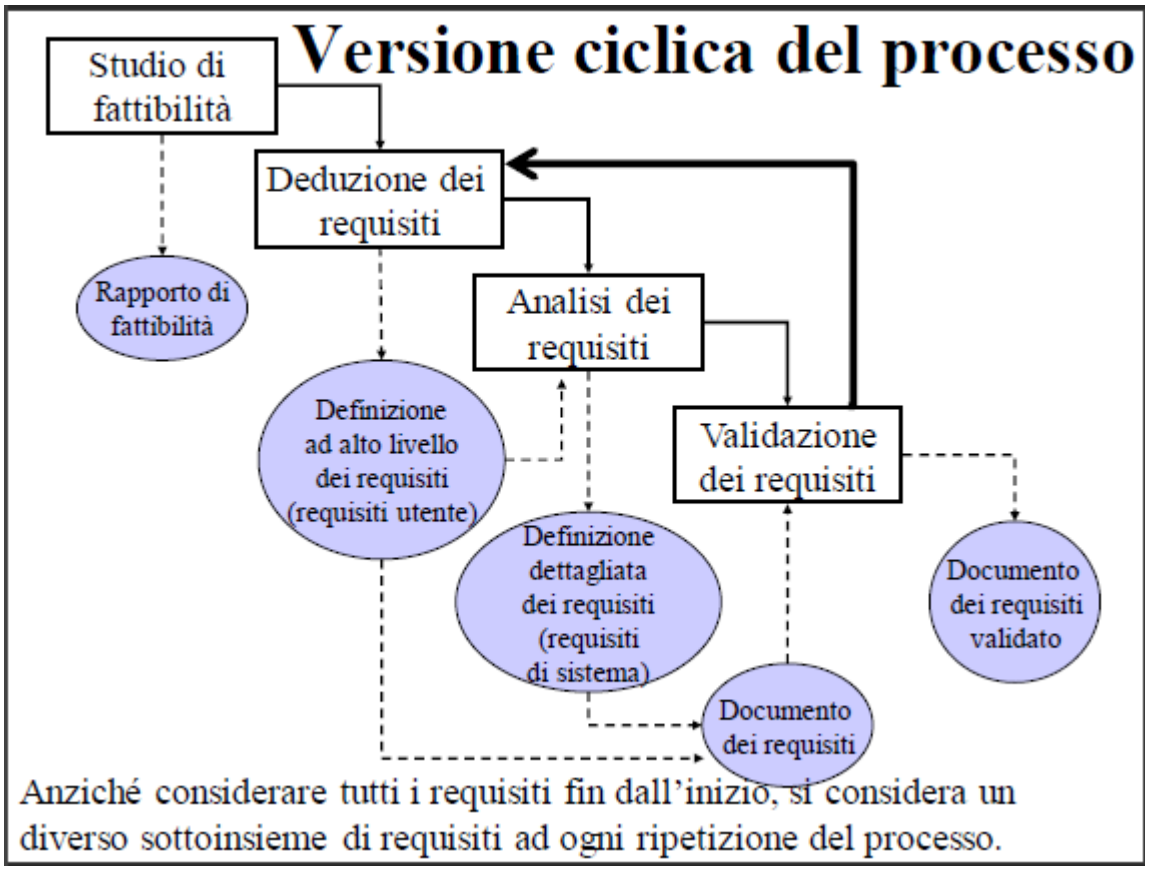
4)Individuare i test-case usati per collaudare il requisito (**tracciabilità dei test –** riguarda la fase di collaudo)

Tracciabilità dei requisiti
Quando si modifica un requisito bisogna valutarne l'impatto sul resto della specifica e sul resto del processo.

Matrice di tracciabilità
Le matrici di tracciabilità servono per collegare i requisiti alle fonti, ai requisiti dipendenti, ai componenti, ai test-case.

Tecniche di validazione
Revisione: un gruppo di revisori controlla i requisiti
1)Controllo manuale della specifica dei requisiti in cerca di anomalie e omissioni
2)I revisori appartengono al team di sviluppo o sono collaboratori del committente
Costruzione di prototipi: uno o più prototipi sono generati e mostrati al committente e agli utenti per verificare la comprensione dei requisiti
Nota: I problemi nei requisiti si risolvono ripetendo le fasi precedenti alla validazione ed è raro scoprire tutti i problemi dei requisiti nella fase di validazione dei requisiti.

Schema processo di specifica con ritorno alla seconda fase se la validazione dei requisiti non va a buon fine.



Fase di progettazione

Tale fase si suddivide in quattro fasi:

1)Progettazione architetturale: tale progettazione si suddivide in altre 4 fasi:

- **1)Strutturazione del sistema:** che definisce sottosistemi e moduli
- **2)Deployment**: distribuzione dei componenti sui dispositivi.
- **3)Metodo di controllo:** definire quali componenti invocano le operazioni e quali le eseguono.
- **4)Modellazione del comportamento**: definire le interazioni tra componenti allo scopo di svolgere una certa funzione.

Per far ciò si usano diagrammi di sequenza con UML.

2)Progettazione delle strutture dati: in cui si definiscono le strutture dati che mantengono i dati del sistema e per far ciò si usano diagrammi delle classi con UML.

3)Progettazione degli algoritmi: in cui si definiscono gli algoritmi per le funzioni del sistema. Per far ciò si usano diagrammi di flusso o diagramma delle attività con UML.

4)Progettazione della Graphical User Interface (GUI): in cui si definisce attraverso degli "sketch" come sarà l'interfaccia grafica.

Descriviamo ora le quattro fasi della progettazione architetturale:

1)Strutturazione del sistema: Il sistema può essere strutturato in vari sottosistemi che tra di loro interagiscono. Ogni sottosistema è composto da dei moduli che sono eseguiti su una certa macchina (deployment).

Un sottosistema è una parte del sistema dedicata a svolgere una certa attività.

Un modulo invece è una parte di un sottosistema dedicata a svolgere particolari funzioni legate all'attività del sottosistema.

In pratica, un sistema è composto da sottosistemi mentre il sottosistema è composto da moduli.

In genere i sottosistemi sono tre:

- **1)Presentazione:** che rappresenta l'interfaccia grafica.
- **2)Elaborazione:** che elabora i dati di input e produce dati di output.
- **3)Gestione dei dati (database):** che si occupa si aggiungere, modificare, rimuovere e ricercare dei dati.

2)Deployment: è la distribuzione dei componenti in vari dispositivi hardware (pc, server, …). Ci sono vari tipi di deployment:

- **1) 1-tier:** i tre strati del sistema sono concentrati su **1** dispositivo
- **2) 2-tiers:** i tre strati del sistema sono distribuiti su **2** dispositivi.
- **3) 3-tiers:** ogni strato del sistema si trova su un dispositivo dedicato.

2-tiers deployment

2-tiers: i tre strati del sistema sono distribuiti su **2** dispositivi: macchina ‘‘*client*’’ e macchina ‘‘*server*’’.

Due soluzioni:

Thin client

- La macchina “server” si occupa dell’**Elaborazione** e della **Gestione dati**
- La macchina “client” si occupa della **Presentazione**

Fat client

- La macchina “server” si occupa della **Gestione dati**
- La macchina “client” si occupa della **Presentazione** e dell’**Elaborazione**

Thin client e Fat client

1)Thin client: In questo caso la macchina ‘‘server’’ si occupa di tutti i calcoli

La capacità di calcolo della macchina “client” è sprecata

2)Fat client: È presente meno carico sulla macchina “*server*” (una parte del lavoro viene fatta dal lato client).

Nota l’aggiornamento del software di elaborazione deve essere fatto su tutte le macchine “client”.

Thin/Fat client con codice mobile

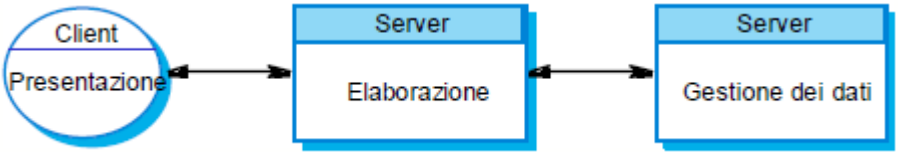
1)Thin client: la macchina ‘‘client’’ prima di entrare in funzione, scarica dalla macchina ‘‘server’’ il software di presentazione.

2)Fat client: la macchina ‘‘client’’ prima di entrare in funzione, scarica dalla macchina ‘‘server’’ il software di presentazione ed elaborazione.

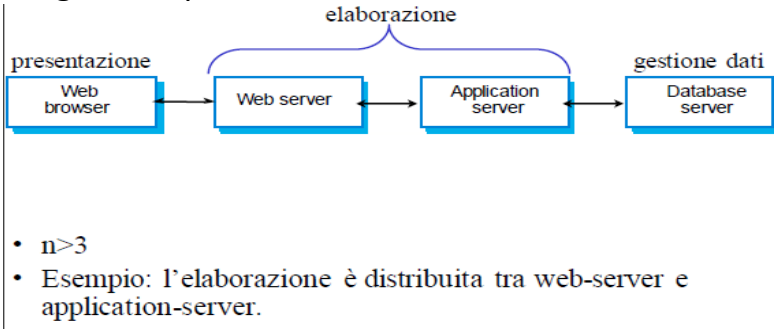
In caso di modifiche è sufficiente aggiornare il software di presentazione (ed elaborazione) sulla macchina ‘‘*server*’’, anziché su tutte le macchine ‘‘*client*’’.

3-tiers deployment

I tre strati del sistema sono eseguiti su tre macchine separate come riportato in figura:



n-tiers deployment
in questo caso i tre sottosistemi sono divisi non più come 3-tiers deployment ovvero ogni sottosistema in una macchina bensì, per esempio, un sottosistema può essere eseguito su più macchine.



3)Metodo di controllo

Un componente fornisce servizi (ovvero procedure, funzioni, metodi, esecuzione di query) ad altri componenti.

1)Interfaccia: insieme di operazioni che il componente mette a disposizione di altri componenti. Essa è condivisa con i componenti che lo invocano.
2)Corpo: parte interna del componente. È l'implementazione dell'interfaccia. Non è conosciuto dagli altri componenti (è ''segreto'').
3)Information hiding: È la separazione tra interfaccia e corpo.
Un **componente** deve nascondere i propri algoritmi e dati ai componenti esterni. Un componente deve essere accessibile solo attraverso la sua interfaccia (insieme di operazioni pubbliche). Modificare l'implementazione interna di un modulo non deve avere effetti sul resto del sistema.

Ci sono diversi stili di controllo:
1)Controllo centralizzato: un componente (controllore) ha il compito specifico di controllo ovvero attiva, disattiva e coordina gli altri componenti del sistema a seconda dello stato del sistema. Il controllore esegue un control loop periodico in cui controlla se dei componenti hanno prodotto dati da elaborare, attiva o disattiva componenti a seconda dei dati raccolti, passa i dati ai componenti per l'elaborazione e raccoglie i risultati.

2)Controllo basato su eventi: In questo caso non c'è un controllore e ogni componente si attiva autonomamente come reazione ad eventi esterni determinati da altri componenti o dall'ambiente. Quindi ogni componente è il controllore di sé stesso.

Un evento si può verificare in qualunque momento ed inoltre, gli eventi possono essere periodici o aperiodici.

In questo sistema di controllo è presente **un gestore degli eventi detto broker** il cui compito è quello di rilevare l'evento e se lo notifica a tutti i componenti viene detto **broadcast non selettivo** mentre se lo notifica ai componenti interessati viene detto **broadcast selettivo.** Il gestore degli eventi mantiene un registro con gli eventi e i corrispondenti componenti di interesse. I componenti possono comunicare al gestore quali sono gli eventi di loro interesse

3)Controllo call-return (top-down): un componente principale attiva altri componenti che a loro volta ne attivano altri.
Top-down: il controllo passa dall'alto verso il basso.
Call: viene eseguito il componente principale che richiama le operazioni dei componenti al 2° livello, i quali a loro volta richiamano i componenti al 3° livello, e così via.
Return: ogni operazione ritorna un risultato al livello superiore, al componente che l'aveva invocata.
Un componente collocato su un certo livello è il controllore dei componenti sul livello sottostante.

4)Controllo client-server: un componente client attiva uno o più componenti server.
In questo controllo l'architettura è composta da:

- Insieme di componenti server che forniscono servizi
- Insieme di componenti client che richiedono servizi ai moduli server

Protocollo request/reply:
Request: Un componente client richiede un servizio ad un componente server attraverso una chiamata di procedura
Reply: Il componente server invia una risposta al modulo client

Ogni componente client è controllore dei componenti server che può invocare. Un client può comunicare con più server.

4)Modellazione del comportamento ad oggetti

I componenti del sistema sono considerati come **oggetti** che interagiscono.
Un oggetto è composto da:

1)Attributi: definiscono lo stato dell'oggetto

2)Operazioni: operazioni che l'oggetto può compiere

Gli oggetti comunicano attraverso lo scambio di messaggi (invocazioni di operazioni e valori di ritorno). Il modello a oggetti può essere mappato in un linguaggio object-oriented durante la fase di implementazione.
UML: diagramma di sequenza

Fase di collaudo
Quando si arriva in questa fase il sistema è stato implementato e si ricercano e si correggono i difetti (bachi, anomalie, omissioni)

Nella fase di collaudo si controlla che:
→Il prodotto implementato realizzi ogni servizio senza malfunzionamenti (**Verifica**)
Lo scopo della verifica è scoprire bachi (errori, cause di crash, corruzione di dati)

Il baco (difetto di programmazione) causa il malfunzionamento (comportamento imprevisto del sistema)

→Il prodotto implementato soddisfi i requisiti del committente (**Validazione**)
Lo scopo della validazione è scoprire anomalie o omissioni.

–**Anomalie**: servizio non fornito nel modo previsto nella specifica
–**Omissioni**: servizi previsti nella specifica, ma non implementati

I possibili bachi che si possono trovare nella fase di verifica sono:
1)Human Error: La causa di un baco è un errore umano (es.: errore di battitura, errore logico)
2)Fault (baco): È un difetto del programma dovuto a un errore (es.: simbolo sbagliato all'interno di un'operazione algebrica)
3)Failure (malfunzionamento): È la manifestazione visibile della presenza di un baco (es.: divisione per zero causa il crash) che causa un comportamento imprevisto.

Tecnica statica (ispezione): Basata sulla lettura del codice e della documentazione (documento dei requisiti, modelli di progettazione). Con questa tecnica il sistema non viene messo in esecuzione.
Tecnica dinamica (testing): Con questa tecnica il sistema viene messo in esecuzione e si osserva come il sistema si comporta quando elabora dei test-case.

Motivazione del perché viene fatta l'ispezione invece di fare subito il testing
(Perché no il testing)
1)Il testing può essere costoso e serve un numero elevato di test-case e di esecuzioni per collaudare ogni parte del codice.
2)Ogni test-case può scoprire un difetto (o alcuni difetti).
3)Un difetto può nascondere altri difetti, infatti, una volta corretto un difetto, non sappiamo se i successivi comportamenti anomali del sistema dipendono da effetti collaterali introdotti correggendo il difetto, o da altri difetti.
(Perché si l'ispezione)
1)L'ispezione è meno costosa (non bisogna eseguire codice).
2)Non si può testare una versione incompleta (non eseguibile), ma si può ispezionarne il codice.
3)Alcuni requisiti non funzionali si possono collaudare solo attraverso l'ispezione come, ad esempio, la portabilità e manutenibilità dipendono dal tipo di linguaggio e dalla strutturazione del codice.
(perché no l'ispezione)
L'ispezione non è in grado di collaudare altri requisiti non funzionali:
1)Efficienza, affidabilità, usabilità, …
2)La validazione di questi requisiti richiede l'esecuzione del prodotto

La strategia da adottare è fare prima l'ispezione del codice, poi il testing del prodotto. Infatti, più difetti vengono scoperti durante l'ispezione, meno test-case dovranno essere definiti ed eseguiti durante il testing.

Ispezione del sistema
Un team analizza il codice e segnala possibili difetti. Il codice viene modificato se si scoprono difetti. Prima di iniziare una ispezione del codice bisogna avere i seguenti requisiti:
1)Il codice deve essere supportato da altri documenti (documento dei requisiti, modelli di progettazione).
2)Il codice deve essere sintatticamente corretto (compilabile).
3)Serve una check-list che indica i possibili difetti da investigare.

Ruoli nel team di ispezione
- **Autori**: programmatori del codice; correggono i difetti rilevati durante l'ispezione.
- **Ispettori**: trovano difetti.
- **Moderatore**: gestisce il processo di ispezione.

Processo di ispezione

1)Pianificazione: il moderatore
–Seleziona gli ispettori
–Controlla che il materiale (codice, documentazione) sia completo
2)Introduzione: il moderatore organizza una riunione preliminare con autori e ispettori
–Gli autori descrivono lo scopo del codice
–Si visiona il materiale
–Si stabilisce la checklist
3)Preparazione individuale: gli ispettori studiano il materiale e cercano difetti nel codice in base alla check-list e all'esperienza personale
4)Riunione di ispezione: gli ispettori indicano i difetti individuati
5)Rielaborazione: il programma è modificato dall'autore per correggere i difetti
6)Prosecuzione: il moderatore decide se è necessario un ulteriore processo di ispezione

ANALISI STATICA DEL CODICE: Strumenti CASE supportano l'ispezione del codice eseguendo sul codice le seguenti analisi:
1)Analisi del flusso di controllo
- cicli con uscite multiple (es.: break)
- salti incondizionati all'interno di un ciclo (goto)
- codice non raggiungibile come ad esempio:
  - –Delimitato da istruzioni di salto incondizionato (goto)
  - –Vincolato ad una condizione sempre falsa

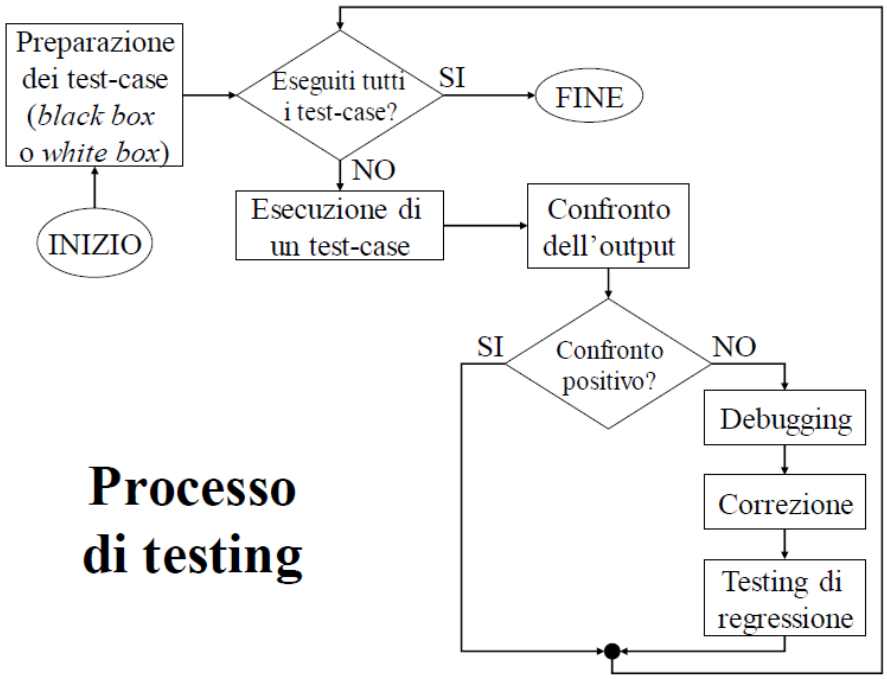
2)Analisi dell'uso dei dati:
- variabili non inizializzate prima di essere usate
- variabili scritte, ma non lette
- variabili dichiarate ma non usate
- violazione dei limiti di un array
- condizioni che danno sempre lo stesso valore Booleano

3)Analisi delle interfacce:
- consistenza delle dichiarazioni di metodi, funzioni, procedure come ad esempio:
  - –Errore nel tipo di parametri, nel numero di parametri
- Metodi, funzioni, procedure dichiarate, ma mai invocate
- Risultati di funzioni non usati

4)Analisi della gestione della memoria:
- Puntatori non assegnati
- Errori nell'aritmetica dei puntatori

L’analisi statica precede l’ispezione del codice fornendo informazioni utili all’individuazione dei difetti però tale analisi non è sufficiente.
–Esempio: individua inizializzazioni mancanti, ma non inizializzazioni errate.
–Esempio: individua il passaggio di parametri di tipo errato, ma non il passaggio del parametro errato.

Fase di testing



Processo di testing

1)Preparazione dei test-case (‘‘*black box*’’ o ‘‘*white box*’’)
- dati di input del test
- dati di output attesi

2)Esecuzione di un test-case fornendo al sistema i dati di input del test-case
3)Confronto dei risultati di output ritornati dal sistema con i dati di output attesi
- Se i dati di output attesi e quelli ritornati non corrispondono, allora c’è la presenza di un difetto e si eseguono le fasi successive

→**Debugging**:
1)Si controlla l’esecuzione del programma, istruzione per istruzione
2)Dopo l’esecuzione di ogni istruzione si controllano i valori delle variabili
3)Si scopre la posizione dell’errore
→**Correzione** dell’errore in cui si modifica il codice per eliminare il difetto
→**Testing di regressione**:
1)Si ripete l’ultimo test-case per verificare che le modifiche al codice abbiano effettivamente corretto l’errore.
2)Si ripetono i test-case precedenti per verificare che le modifiche non abbiano introdotto altri difetti: effetto collaterale.
NOTA BENE: Il processo di testing viene ripetuto per ogni test-case

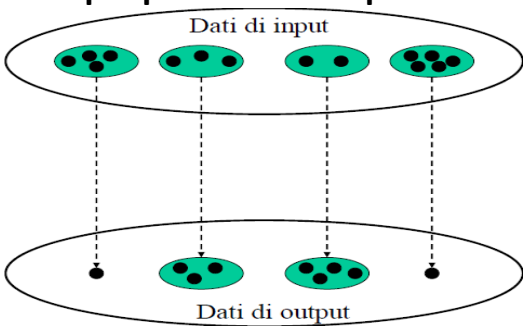
Test-case
Un test-case è composto da:
–Dati di input
–Dati di output attesi
I dati di input possibili potrebbero essere infiniti quindi i possibili test-case potrebbero essere infiniti.
Il sistema può essere testato un numero finito di volte e serve un criterio per scegliere un numero finito di test-case che permetta un testing efficace del sistema e esistono due tipi di approcci: **Black box** o **White box**.

Approcci per scegliere i test-case
1)Black box testing: Il sistema è visto come una scatola non trasparente:
La scelta dei test-case è basata sulla conoscenza di quali sono i dati di input e quali sono i dati di output.
•Si considerano solo gli input e gli output della funzione da testare e non si considerano il modo in cui i dati di input vengono trasformati in dati di output, quindi, non sono considerate i metodi usati o meglio gli algoritmi usati.

2)White box (o glass-box) testing: Il sistema è visto come una scatola trasparente.
La scelta dei test-case è basata sulla struttura del codice.
•In questo caso è visibile il modo in cui i dati vengono elaborati.

Black Box: partizioni di equivalenza
•Dato l'insieme dei dati di input e l'insieme dei dati di output, una **partizione di equivalenza** è un sottoinsieme dei dati di input per cui
–Il sistema produce sempre lo stesso dato di output
–Oppure il sistema produce dati di output che appartengono sempre ad un determinato sottoinsieme
•Normalmente, il sistema si comporta nello stesso modo per i dati di input che appartengono alla stessa partizione.
•Si può fare black box testing usando delle partizioni di equivalenza:
–Si suppone che se un test è negativo/positivo per alcuni elementi della partizione, allora il test è negativo/positivo per tutti gli elementi della partizione.

Esempio partizioni di equivalenza



Individuare partizioni e test-case

1)Si identificano i possibili dati di output e se sono numerosi, si distribuiscono in sottoinsiemi.

2)Per ogni dato di output o sottoinsieme di dati di output si individuano una o più partizioni di equivalenza tra i dati di input

3)Da ogni partizione si scelgono un numero finito di test-case

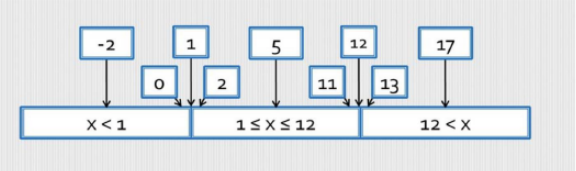
•**Boundary Value Analisys (BVA):** si scelgono test-case con dati di I/O al **confine** delle partizioni di equivalenza.

•Maggiore è il partizionamento dei dati, maggiore sarà il numero di test-case, e maggiore sarà l'efficacia del testing.

Esempio: giorni del mese

Funzione che dato un mese espresso come numero (1..12) restituisce il numero di giorni in quel mese.

- Insieme dei possibili dati di output: {31, 30, 28, -1}
- Insieme dei possibili dati di input: tutti i valori *int*
- Partizioni di output: {31, 30, 28}, {-1}
- Partizioni di input:



- Test-cases per output -1: (-2,-1), (0,-1), (13,-1), (17,-1)
- Test-cases per output in {31, 30, 28}: (1,31), (2,28), (5,31), (11,30), (12,31)

White box testing

Il collaudatore può vedere il codice. I test-case sono ricavati dall'analisi del codice e l'obbiettivo è testare ogni parte del codice:

–Ogni istruzione nel programma deve essere eseguita almeno una volta durante il testing

–Ogni condizione nel programma deve essere eseguita sia nel caso sia vera, sia nel caso sia falsa

White box: flow-graph

Flow graph (grafo di flusso): È un grafo che rappresenta i possibili cammini nel codice

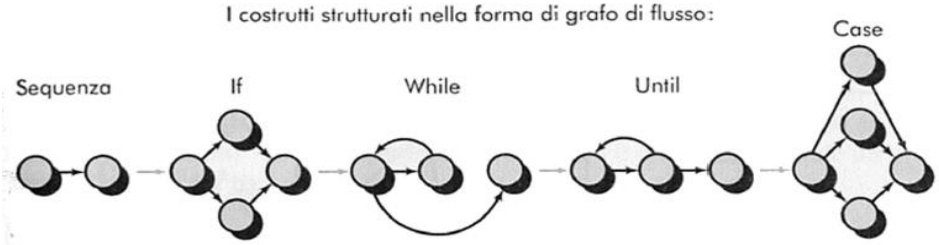
–Un nodo rappresenta:

- •un'istruzione
- •un gruppo di istruzioni in sequenza
- •una condizione

–Gli archi rappresentano il flusso del controllo:

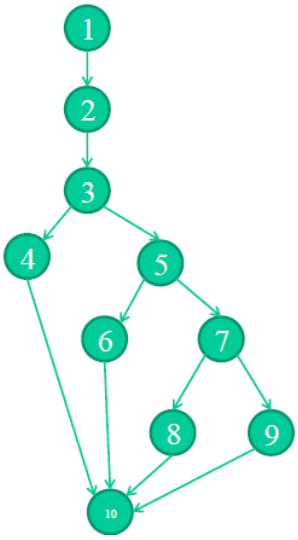
- •Il flow graph può essere ricavato in modo manuale o automatico (strumento CASE)

I COSTRUTTI USATI ALL'INTERNO DEL FLOW GRAPH



ESEMPIO DELL'UTILIZZO DEL FLOW GRAPH

```
// funzione che dato il mese (1..12) restituisce il numero di giorni nel mese
1  int giorni_mese(int mese) {
2     int giorni;
3     if (mese<1 || mese >12)
4           giorni = -1;
5     else if (mese==2)
6           giorni=28;
7     else if (mese==4 || mese==6 || mese==9 || mese==11)
8           giorni=30;
9     else giorni=31;
10    return giorni; }
```



Cammini indipendenti

Nel codice, un cammino si dice **indipendente** se introduce almeno una nuova sequenza di istruzioni o una nuova condizione.

Nel flow-graph corrispondente, un cammino è **indipendente** se attraversa almeno un arco non ancora percorso nei cammini precedenti.

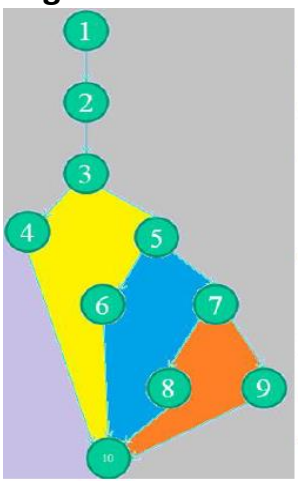
Complessità ciclomatica: Il numero di cammini indipendenti equivale alla Complessità Ciclomatica (CC) del flow-graph

CC = #archi - #nodi + 2

CC = #nodi_predicato + 1

CC = #regioni

Regione: area delimitata da archi e nodi oppure area esterna al grafo



CC = #archi - #nodi + 2 = 12 – 10 + 2 = **4**

CC = #regioni = **4**

CC = #nodi_predicato + 1 = 3 + 1 = **4**

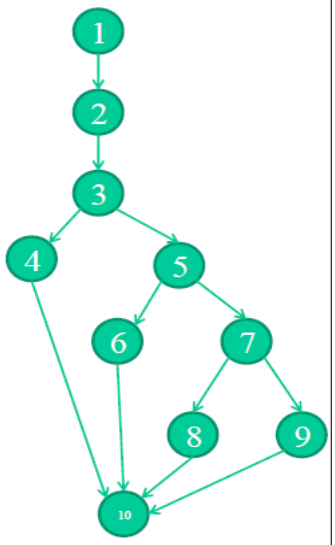
3 cammini indipendenti:

1) 1, 2, 3, 4, 10
2) 1, 2, 3, 5, 6, 10
3) 1, 2, 3, 5, 7, 8, 10
4) 1, 2, 3, 5, 7, 9, 10

Esempio di flow-graph e complessità ciclomatica (1)

```
1  int giorni_mese(int mese) {
2     int giorni;
3     if (mese<1 || mese >12)
4           giorni = -1;
5     else if (mese==2)
6           giorni=28;
7     else if (mese==4 || mese==6 || mese==9 || mese==11)
8           giorni=30;
9     else giorni=31;
10    return giorni; }
```

Test cases



Cammino 1: 1, 2, 3, 4, 10
Input: 0 **Ouput atteso**: -1

Cammino 2: 1, 2, 3, 5, 6, 10
Input: 2 **Ouput atteso**: 28

Cammino 3: 1, 2, 3, 5, 7, 8, 10
Input: 4 **Ouput atteso**: 30

Cammino 4: 1, 2, 3, 5, 7, 9, 10
Input: 1 **Ouput atteso**: 31

La complessità ciclomatica è il numero minimo di test-case richiesti per eseguire almeno una volta ogni parte del codice.

Stabilendo un test-case per ogni path indipendente, si esegue ogni parte del codice almeno una volta.

Dynamic program analyzer: sono strumenti CASE che, dato un test-case, indicano:

1)quali parti del codice (flow-graph) sono state interessate da un test-case

2)quali parti sono ancora da testare

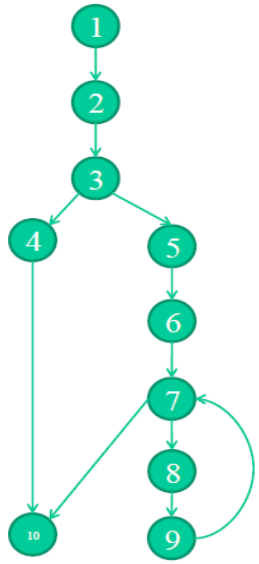
Essi sono utili quando il flow graph è complesso ad esempio:

plug-in *EclEmma* per *Eclipse*

Esempio di flow-graph e complessità ciclomatica (2)

```
// funzione che calcola il fattoriale (n!) di un numero n>=0
1  int fattoriale(int n) {
2     int i, fatt;
3     if (n<0)
4        fatt = -1;
5          else {    fatt=1;
6                    i=n;
7                    while (i>1) {
8                        fatt=fatt*i;
9                        i--; } }
10    return fatt; }
```

Test cases



Cammino 1: 1, 2, 3, 4, 10
Input: -1 **Ouput atteso**: -1

Cammino 2: 1, 2, 3, 5, 6, 7, 10
Input: 0 **Ouput atteso**: 1

Cammino 3: 1, 2, 3, 5, 6, 7, 8, 9, 7
Input: 3 **Ouput atteso**: 6

Tipi di componenti da integrare nel sistema

Componenti sviluppati ad-hoc per il sistema.

–sono tipicamente implementati in parallelo.

Componenti commerciali (COTS - Commercial Off-The-Shelf) già disponibili.

–Il costo è inferiore rispetto ad un componente ad-hoc.

Componenti realizzati per sistemi precedenti.

–Costo quasi nullo.

Approcci di integrazione dei componenti

1)Integrazione: composizione dei componenti in un unico sistema.

2)Approccio incrementale: i componenti sono integrati uno alla volta.

–La consegna dei componenti deve essere coordinata, ma non contemporanea.

3)Approccio "big bang": i componenti sono tutti integrati contemporaneamente.

–I componenti devono essere completati entro lo stesso termine temporale.

Testing con integrazione incrementale

Il testing riguarda ciascun componente e la sua integrazione nel sistema.

–**Testing dei componenti**: testing di ogni componente (ad-hoc) considerato in modo isolato.

–**Testing di integrazione**: si testa il sistema costituito dai componenti testati ed integrati finora.

- Viene ripetuto ogni volta che un componente viene integrato

–**Release testing**: testing del sistema costituito da tutti i componenti

- Prodotti customizzati: test di accettazione
- Prodotti generici: alpha testing, beta testing

Testing dei componenti: Riguarda i singoli componenti

Il test di un componente:

1)Può essere preceduto dall'**ispezione** del codice

- Il codice di un componente non è di solito troppo lungo

2)Può essere di tipo **white-box**

- Il codice di un componente non è di solito troppo lungo

3)Ha lo scopo di **verifica** (ricerca dei bachi)

- Di solito non si può fare la validazione sulla base di un singolo componente

4)Viene svolto di solito dal programmatore del componente

Testing di integrazione

Viene svolto quando si integrano due o più componenti.

–Lo scopo del test di integrazione è verificare l'interfacciamento corretto dei componenti.

1)Le funzioni di un componente devono essere invocate nel modo corretto.

2)Le funzioni di un componente devono ritornare i risultati attesi.

–Si applicano i test-case applicati nei test di integrazione precedenti (test di regressione).

•L'aggiunta di nuovi componenti può creare problemi nel funzionamento di quelli integrati e testati precedentemente

–Si applicano nuovi test-case

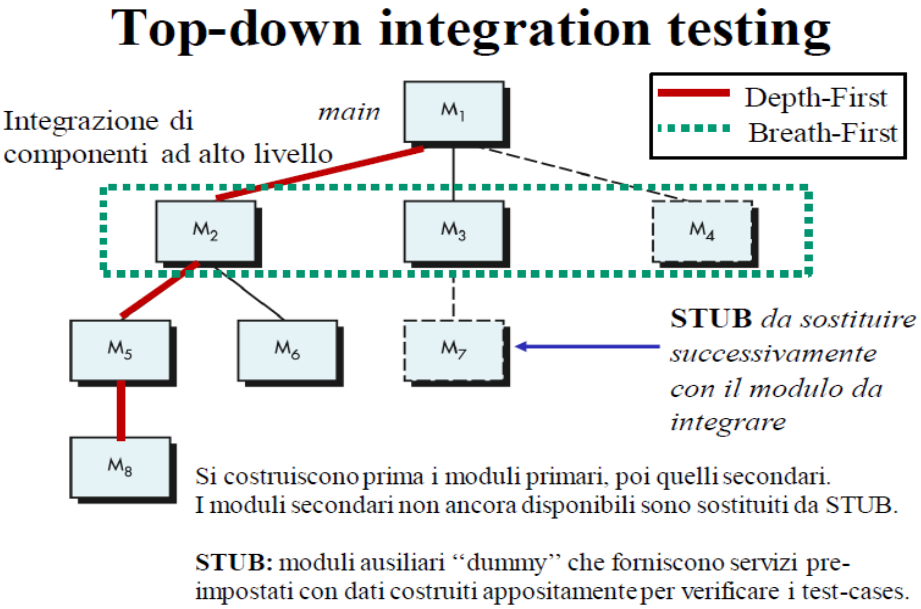
NOTA BENE: Quando si presenta un difetto, è più probabile che esso riguardi l'ultimo componente integrato.

Il test di integrazione riguarda la verifica e può riguardare la validazione se sono già stati integrati i componenti necessari per un certo requisito.

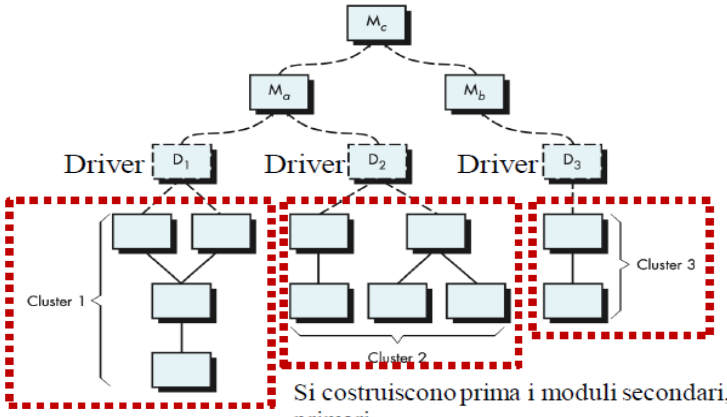
I tipi di testing sono:

1)White box se pochi componenti sono stati integrati finora

2)Black box se molti componenti sono stati integrati finora



Bottom-up integration testing



Integrazione di componenti a basso livello

Si costruiscono prima i moduli secondari, poi quelli primari.
I moduli primari non ancora disponibili sono sostituiti da DRIVER.
DRIVER: test program che invoca il modulo secondo i dati selezionati per i test-case.

Release testing

- Riguarda il sistema completo (release)
- Riguarda la **validazione** (validation testing)
- Testing di tipo **black box**
- Due tipi di release testing
  - –Sistemi customizzati: test di accettazione
  - –Sistemi generici: alpha e beta testing

Sistemi customizzati: test di accettazione

1)Il committente controlla il soddisfacimento dei requisiti
2)Si effettuano eventuali correzioni

Queste due fasi si ripetono finché il committente non è soddisfatto; dopodiché il sistema viene effettivamente consegnato.

Sistemi generici

1)Alpha testing

- La versione “Alpha” del sistema viene resa disponibile ad un gruppo di sviluppatori (tester) o utenti esperti, per il collaudo
- La versione “Alpha” viene collaudata ed eventualmente corretta
- Si ripete l’Alpha testing oppure si genera la versione “Beta”

2)Beta testing

- La versione “Beta” del sistema viene resa disponibile (via web) ad un gruppo di potenziali clienti per il collaudo
- La versione “Beta” viene collaudata dai clienti ed eventualmente corretta
- Si ripete il Beta testing oppure il prodotto viene messo sul mercato

Testing con integrazione "big-bang"

I componenti sono integrati tutti insieme in una sola volta. Si effettuano:

1)Test dei componenti (prima dell'integrazione)

2)Release testing (dopo l'integrazione)

•Non si effettua testing di integrazione

La difficoltà nel localizzare un eventuale difetto dopo l'integrazione è capire quale componente riguarda. (*''Integration hell''*)

Stress testing

Serve per verificare:

1)efficienza (tempo di risposta, memoria occupata, numero di job processati, ecc.): il sistema deve elaborare il carico di lavoro previsto

2)affidabilità: il sistema non deve fallire (andare in crash)

Viene svolto quando il sistema è completamente integrato (efficienza e affidabilità sono proprietà complessive ed emergenti): riguarda software e hardware.

In questo caso si eseguono test in cui il carico di lavoro è molto superiore a quello previsto normalmente.

–Se il sistema "resiste" a carichi di lavoro superiori al previsto, dovrebbe resistere a carichi di lavoro normali

–Se il sistema non "resiste" è possibile individuare difetti non emersi dagli altri test come per esempio: corruzione dei dati, perdita dei servizi, …

Test di usabilità

•Vari tipi di utente provano ad utilizzare il sistema per la prima volta.

–Utenti non esperti

–Utenti che hanno già usato sistemi simili

–Utenti con competenze tecniche

•Si valuta la facilità con cui riescono ad usare il sistema e si raccolgono commenti, critiche e suggerimenti dagli utenti.

Altri tipi di testing

•**Recovery testing:** valida requisiti come

Fault tollerance, Recoverability, Mean Time to Repair

•**Security testing:** simulazione di attacchi dall'esterno

•**Deployment testing:** verifica dell'installazione sui dispositivi

Back-to-back testing

- Si usa quando varie versioni del sistema sono disponibili
  - –Prototipi
  - –Versioni del sistema per sistemi operativi diversi
  - –Versioni del sistema per tipi di hardware diversi
  - –Nuove versioni con funzioni in comune con le precedenti
- Si effettua lo stesso test su tutte le versioni e si confrontano gli output delle varie versioni
  - –Se l'output non è sempre lo stesso, c'è la presenza di difetti in qualche versione
  - –Se l'output è sempre lo stesso, è probabile che sia corretto oppure tutte le versioni potrebbero avere lo stesso difetto

Fase di Manutenzione

La manutenzione riguarda tutte le modifiche fatte al sistema **dopo la consegna.**
È un processo ciclico che dura dalla consegna alla dismissione del sistema.
La fase di manutenzione permette al sistema di "sopravvivere". Un sistema va aggiornato nel tempo oppure perde progressivamente di qualità, utilità e valore economico.

Tipi di manutenzione

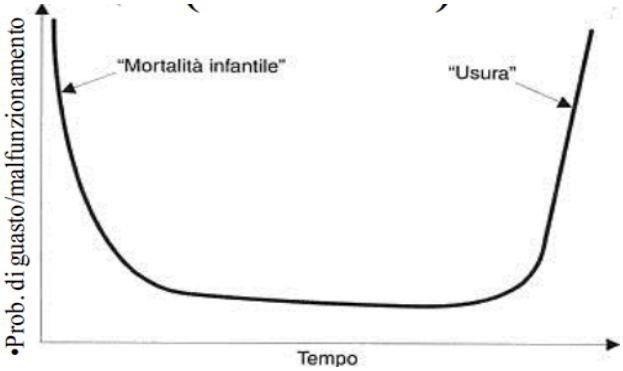
1)manutenzione correttiva: correzione di difetti non emersi in fase di collaudo. Ci sono vari tipi di difetti:

1)**Difetti di implementazione**: sono i meno costosi da correggere
2)**Difetti di progettazione**: sono più costosi perché richiedono di modificare vari componenti del sistema
3)**Difetti di specifica**: sono i più costosi da correggere: potrebbe essere necessario riprogettare il sistema

2)manutenzione adattativa: adattamento del sistema a cambiamenti di piattaforma (nuovo hardware o nuovo sistema operativo)
3)manutenzione migliorativa: aggiunta, cambiamento o miglioramento di requisiti funzionali e non funzionali, secondo le richieste del committente/cliente/utente o in base alle tendenze di mercato

Nota bene: Il costo della manutenzione è pari o superiore al costo di sviluppo.
Inoltre, è la manutenzione migliorativa la più costosa tra le varie manutenzioni.

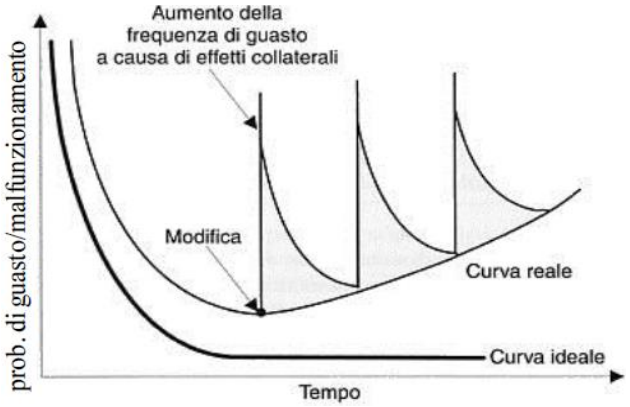
Curva dei guasti dell'hardware



1)"mortalità infantile": È il periodo in cui si possono rappresentare difetti non scoperti nel collaudo.

2)Cause dell'"usura": L'usura può avvenire per svariati motivi come per esempio deterioramento dei materiali, polvere, vibrazioni, fattori ambientali, temperatura.

Curva dei guasti del software



Curva ideale: nel caso si richieda solo manutenzione correttiva, infatti, in questo caso non c'è usura per il software.

Curva reale: causata da modifiche di manutenzione migliorativa.

Ogni modifica può introdurre difetti (effetti collaterali) e quindi successive richieste di manutenzione correttiva.

Fattori che influenzano il costo di manutenzione

1)Dipendenza dei componenti: la modifica di un componente potrebbe avere ripercussioni sugli altri componenti.

2)Linguaggio di programmazione: i programmi scritti con linguaggi ad alto livello sono più facili da capire e quindi da mantenere.

3)Struttura del codice: il codice ben strutturato e documentato rende più facile la manutenzione.

4)Collaudo: una fase di collaudo approfondita riduce il numero di difetti scoperti successivamente alla consegna.

5)Qualità della documentazione: una documentazione chiara e completa facilita la comprensione del sistema da mantenere.

6)Stabilità dello staff: i costi si riducono se lo staff che ha sviluppato il prodotto è lo stesso che lo mantiene.

7)Età del sistema: il costo di manutenzione tende a crescere con l'età del sistema. Un esempio è dato dall'uso di linguaggi superati (COBOL, FORTRAN, …)

8)Stabilità del dominio dell'applicazione: se il dominio subisce variazioni, il sistema deve essere aggiornato (es.: programmi gestionali)

9)Stabilità della piattaforma: il cambiamento della piattaforma (hardware o sistema operativo) può richiedere la manutenzione del software; è raro che la piattaforma non cambi durante la vita del prodotto software.

Processo di manutenzione (le 5 fasi)

1)Identificazione dell'intervento: arrivo di una richiesta di modifica (nuovi requisiti, correzione di difetti, adattamenti, …)

2)Analisi dell'impatto:

- –individuare requisiti, componenti, operazioni e test-case influenzati dalla modifica.
- –Individuare le possibili soluzioni e valutare tempi, costi e benefici di ognuna.

3)Pianificazione della release

- –Selezionare la soluzione migliore.
- –Accettare o rifiutare la modifica.

4)Realizzazione della modifica: se la modifica è accettata,

- –Se necessario, aggiornamento della **specifica** dei requisiti
- –Se necessario, aggiornamento della **progettazione**
- –Sicuramente, aggiornamento dell'**implementazione** (coding)
- –**Collaudo**: test-case relativi alla modifica effettuata + test di regressione (ripetizione dei test-case originali)

5)Rilascio della nuova versione del sistema

Change Request Form (CRF)

È un documento formale che descrive una modifica. Compilato da:

1)Il proponente della modifica (committente, utente finale, sviluppatori):

- Proposta di modifica e priorità

2)Dagli sviluppatori:

- Requisiti, componenti, test-case influenzati dalla modifica
- Valutazione della modifica: benefici, costi, tempi
- Modalità di implementazione

3)Il Change Control Board (CCB): committente, manager, alcuni sviluppatori

- Approvano o rifiutano la proposta della modifica

Manutenzione di emergenza (patch)

1)Si presentano problemi che devono essere risolti in fretta:

1)Errore grave di sistema che non permette la continuazione delle normali operazioni.

2)Malfunzionamento del sistema che causa danni economici.

3)Cambiamenti imprevisti nell'azienda (es.: nuove norme giuridiche).

2)L'urgenza richiede una manutenzione di emergenza:

1)Non si passa attraverso tutte le fasi del processo di manutenzione.

2)Si modifica direttamente il codice, applicando la soluzione più immediata, e si rilascia una versione aggiornata (**patch**).

3)Più tardi si fa una manutenzione ordinaria per la stessa richiesta di modifica e si rilascia un'ulteriore versione.

Versioni e Release

Versione: Istanza del sistema che differisce per qualche aspetto dalle altre istanze.

–Variazione dei requisiti.

–Correzione di difetti.

–Piattaforma.

Release: Una particolare versione che viene distribuita a committente/clienti

–Ci sono più versioni che release.

- Versioni intermedie possono essere create durante lo sviluppo

–Distribuzione via cd-rom/dvd o download.

Una **release** comprende:

1)Programma eseguibile

2)File di configurazione: indicano come la release deve essere configurata per particolari installazioni

3)File di dati: necessari per il funzionamento del sistema

4)Programma di installazione: per installare il sistema su una certa piattaforma

5)Documentazione cartacea, elettronica, on-line.

Identificare numericamente le versioni
1)Numerazione release: 1.0, 2.0, 3.0, …
2)Numerazione versioni: 1.0, 1.1, 1.2, …2.0, 2.1, …
ESEMPIO con 4.7.6
-Il 4 rappresenta il major version
-Il 7 rappresenta la minor version
-Il 6 rappresenta la patches

Attributi di una versione
1)Committente
2)Linguaggio di programmazione
3)Stato dello sviluppo
4)Piattaforma (hardware, sistema operativo)
5)Data di creazione

Identificare le versioni tramite gli attributi
Un numero di versione identifica una versione, ma non fornisce informazioni su di essa.
Identificazione basata su attributi
Una versione è identificata dalla combinazione di alcuni suoi attributi. Inoltre, bisogna garantire **l'unicità** infatti, gli attributi devono essere scelti in modo che ogni versione possa essere identificata in modo univoco.

Configurazione software
È l'insieme di informazioni prodotte da un processo software.
1)Documentazione:
- Documento dei requisiti
- Modelli di progettazione
- Test-case
- Documentazione delle modifiche rispetto alla versione precedente (CRF)
- Informazioni sul committente
- Documenti di gestione (diagrammi di scheduling, assegnazione delle risorse)
- Informazioni su strumenti CASE, librerie, compilatori, … usati

2)Codice
3)Dati

Database delle configurazioni
Questo database contiene le varie configurazioni software corrispondenti alle release di un sistema.

- –Fornisce le informazioni utili quando si deve fare la manutenzione
- –Può essere integrato con il VMS (Git, SVN, ...)
- –Può essere integrato con altri strumenti CASE
  - •Editor dei requisiti
  - •UML editor
  - •IDE
  - •Tool per testing automatico

Sistemi ereditati (Legacy systems)
Un sistema ereditato è un vecchio sistema che deve essere mantenuto nel tempo. La loro tecnologia è obsoleta inoltre sono costosi da mantenere, ma la loro dismissione e sostituzione può essere rischiosa poiché sono sistemi critici che possono contenere informazioni che devono essere conservate o possono essere servizi molto perfezionati durante la vita del sistema.

I problemi che si hanno con i sistemi ereditati sono:

1)Possono essere poco strutturati,
2)Hanno una mancanza di documentazione,
3)È difficile reperire gli sviluppatori originali,
4)Sono sviluppati per vecchie piattaforme (ad esempio, main-frame).

I sistemi ereditati sono difficili da mantenere.

Strategie per i sistemi ereditati
1)Manutenzione ordinaria del sistema.
2)Software re-engineering ristrutturazione dei sistemi ereditati per renderli più mantenibili.
3)Reimplementazione del sistema mantenendo gli stessi requisiti funzionali.
4)Dismissione del sistema ovvero cambiare la propria organizzazione in modo che l'uso del sistema non sia più necessario.
La scelta della strategia dipende dalla qualità del codice del sistema e dalla sua utilità (business value)
Il costo di manutenzione è inversamente proporzionale alla qualità del codice. Quindi più un codice è fatto bene più i costi di manutenzione saranno bassi.

Forward engineering vs re-engineering

1)Forward engineering (ingegneria diretta):
È il processo software (specifica, progettazione, implementazione, collaudo, manutenzione). In questo caso si crea un nuovo prodotto partendo dalla specifica dei requisiti.
2)Re-engineering:
Il nuovo sistema è dato da qualche trasformazione del vecchio sistema, allo scopo di rinnovarlo e aumentarne la manutenibilità. In questo caso i requisiti funzionali non cambiano e non si altera il comportamento del sistema,

Attività di re-engineering:

1)Traduzione del codice
2)Ristrutturazione del codice (refactoring)
3)Ristrutturazione dei dati

Traduzione del codice sorgente
È la traduzione del codice dal linguaggio di programmazione originale ad una versione più recente dello stesso linguaggio o ad un linguaggio differente.
Le ragioni per fare la traduzione sono:

1)Aggiornamento dell'hardware: poiché i compilatori del linguaggio originale potrebbero non essere disponibili per il nuovo hardware.
2)Mancanza di conoscenza del linguaggio: lo staff potrebbe non conoscere il linguaggio originale perché non è più in uso (COBOL)
3)Politiche organizzative: decisione di avere un certo linguaggio come standard.

Ristrutturazione del codice (refactoring)

Le modifiche fatte al sistema nel corso del tempo tendono a rendere il codice sempre meno ordinato e infatti il codice diventa sempre più difficile da leggere e comprendere.

La ristrutturazione cerca di rendere il codice più semplice, leggibile, comprensibile e modulare:

1)Raggruppando delle parti correlate del programma

- il codice relativo allo stesso componente o alla stessa funzionalità, non deve essere sparso ma coeso (es.: nello stesso file o nella stessa directory).

2)Eliminazione delle ridondanze

3)Aggiunta di commenti nel codice

4)Eliminare codice duplicato (tramite funzioni/metodi)

5)Ridurre la lunghezza di una funzione/metodo distribuendo il codice in varie funzioni/metodi

6)Dare nomi significativi a variabili, funzioni, classi, metodi, attributi, per far capire il loro scopo.

7)Fondere classi simili in un'unica classe.

8)Uniformare l'indentazione (spazi, tab, posizione { }) e la posizione dei commenti.

Il codice semplice e uniforme è più leggibile

- La ristrutturazione non modifica il comportamento del codice, ma ne migliora la struttura interna.
- La ristrutturazione avviene ispezionando e editando il codice
- La ristrutturazione è un processo costoso. Deve essere limitata alle parti del codice che effettivamente lo richiedono come ad esempio:

1)Parti in cui si verificano fallimenti: devono essere comprensibili per poter essere corrette

2)Parti che hanno subito molte modifiche: la strutturazione del codice ha subito un degradamento

3)Parti in cui il codice è particolarmente complesso: difficile da leggere

Ristrutturazione dei dati

È un processo di riorganizzazione delle strutture dati e serve per:

1)Cambiamento delle strutture dati

2)Adattare un sistema ad usare un DBMS anziché usare file di dati di formato proprio

3)Concentrare i dati sparsi in vari file in un unico DB

4)Migrazione dei dati da un tipo di DBMS ad un altro

5)Migrazione dei dati in un DBMS più moderno

Reverse engineering

Per sistemi ereditati, la documentazione può essere scarsa o assente, complicando le attività di re-engineering.

Re-engineering può essere supportato dal **Reverse engineering** che è la generazione di documentazione partendo da codice e dati.

1)Input: codice sorgente e dati

2)Output: documentazione

- Diagrammi di progettazione (UML)
- Diagrammi dei dati (E-R)

Il supporto avviene grazie a strumenti CASE come *Visual Paradigm*: Java/C++ UML

Reimplementazione del sistema

La reimplementazione del sistema avviene quando il vecchio sistema è troppo mal strutturato per il re-engineering oppure quando è inevitabile per esempio:

1)Cambiamento radicale del linguaggio di programmazione (es.: da funzionale a object-oriented)

2)Modifiche architetturali sostanziali (es.: da centralizzato a distribuito)

La reimplementazione è la creazione di un nuovo sistema partendo dal vecchio sistema.

1)Il vecchio sistema fornisce la specifica dei requisti funzionali

2)Tutto il resto (req. non funzionali, progettazione, implementazione e collaudo) viene rifatto usando tecnologia aggiornata.

3)Maggiori rischi: introduzione di errori di specifica, progettazione o implementazione

4)Maggiori costi rispetto al re-engineering

5)Il sistema diventa più mantenibile: il sistema è completamente nuovo

GESTIONE DEL PROGETTO

Personaggio chiave a cui ci riferiamo: Manager

Il manager negozia un contratto (tempi e costi) con il committente, è quindi il collegamento tra committente e sviluppatori.

Inoltre, si occupa delle attività di gestione.

Attività di gestione

1)Scrittura proposte: per ottenere un contratto o vincere una gara d'appalto (obbiettivi, metodi, stima preliminare di tempi e costi, …)

2)Pianificazione: task, milestone, risorse, tempi

3)Stima dei costi

4)Gestione dei rischi: previsione di problemi e soluzioni

5)Monitoraggio del processo: si valutano i progressi del processo, e si confrontano con la pianificazione

6)Scrittura e presentazione di rapporti: si informa il committente sull'avanzamento del progetto

7)Revisione del progetto: aggiornamento della pianificazione, della stima dei costi e della gestione dei rischi.

Processo dell'attività di gestione

```
Pianificare il progetto (task, risorse, tempi)
Stimare il costo
While ( (il progetto non è stato completato) and (il progetto non è stato annullato) )
{
    iniziare le attività secondo la pianificazione
    aspettare

    monitorare i progressi del progetto
    revisionare la pianificazione
    revisionare la stima del costo
    if (costi e/o tempi di consegna non possono essere rispettati) then
    {
        rinegozia il contratto con il committente (tempo e costo)
        if (committente non approva) then
            progetto annullato
    }
}
```

Processo software e Gestione dell'attività

Il processo software e la gestione si svolgono in parallelo.

Progetto = processo software + attività di gestione

Scopi delle due fasi che compongono il progetto:

1)processo software: creare un sistema che soddisfi i requisiti

2)gestione dell'attività: il sistema deve essere consegnato nei tempi previsti e deve avere un costo finale non superiore a quello previsto

Un progetto **fallisce** quando

1)Il prodotto è consegnato in ritardo (gestione)

2)Costa più di quanto stimato (gestione)

3)Non soddisfa i requisiti (processo software)

La buona gestione non garantisce il successo di un progetto, ma la cattiva gestione di solito lo fa fallire.

Difficoltà di gestione

1)Risorse limitate: Una risorsa è ciò che viene utilizzato durante un'attività. Degli esempi sono risorse umane, tecnologiche, finanziarie, …

2)Tempo limitato: stabilito con il committente nel contratto

3)Costo limitato: stabilito con il committente nel contratto

4)Irreversibilità: tempo e risorse già impiegate non sono recuperabili

5)Incertezza: le conseguenze delle decisioni non sono certe; si possono verificare problemi (previsti e non previsti)

6)Complessità: bisogna coordinare molte attività distribuite tra vari gruppi di persone

7)Il software è intangibile: non si può vedere materialmente il progresso di un progetto; è difficile stabilire lo stato di avanzamento.

- –Per questo è necessaria la produzione di documentazione

8)I progetti sono "unici": ognuno è diverso dall'altro

- –Non sempre si può fare riferimento all'esperienza fatta in progetti precedenti

Le attività di Pianificazione

1)Scomposizione del processo software in **task** (attività):

1.1) Identificazione delle **dipendenze** tra task

- •Un task può iniziare solo quando un insieme di altri task è stato ultimato
  - –Es.: l'integrazione dei componenti A e B può iniziare solo quando l'implementazione di A e l'implementazione di B sono state entrambe ultimate
- •Task paralleli o sequenziali a seconda delle dipendenze

1.2) Definizione di **milestone** e **deliverable**

2)Assegnazione delle risorse ad ogni task:

- •Persone (con eventuale ricerca di nuovo personale)
- •hardware o software (con eventuale acquisto di macchine o licenze)
- •Budget (denaro spendibile per quel task)

3)Tempistica (Scheduling): stima dei tempi

3.1) Durata di un task: tempo di svolgimento più il tempo di ricerca delle risorse.

- •Regola pratica: fare una stima del tempo come se tutto andasse bene + 30% per rischi previsti + 20% per rischi imprevisti

3.2) Durata del progetto: tempo complessivo

Milestone e deliverable

Per valutare l'avanzamento del processo, si stabiliscono dei momenti precisi in cui determinati risultati intermedi devono essere raggiunti.

1)Milestone: terminazione di un certo insieme di task previsto per una certa data. È un punto di controllo per l'avanzamento del lavoro. In questo caso viene preparato un **rapporto** sullo stato di avanzamento del progetto che fornisce le informazioni per verificare il rispetto della pianificazione.

2)Deliverable: particolare milestone i cui risultati devono essere notificati al committente

Frequenza delle milestone: se troppo frequenti, causano uno spreco di tempo nella preparazione di rapporti; se poco frequenti, non permettono di valutare l'avanzamento del processo.

Esempio di scheduling

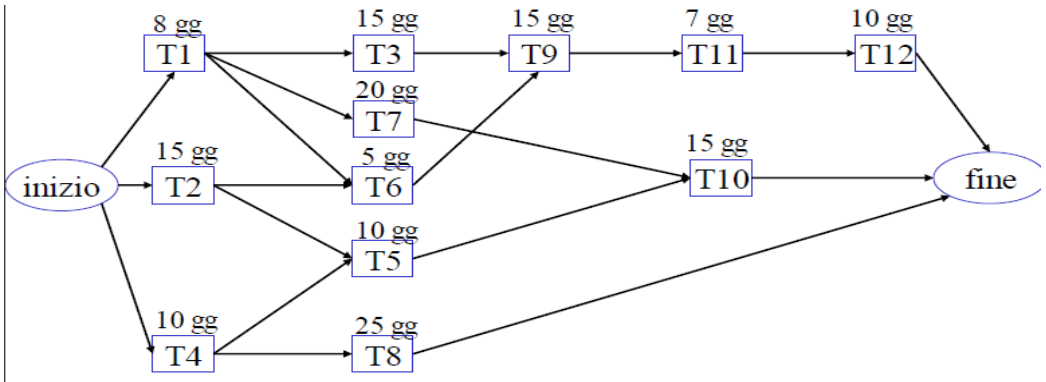
Activity	Duration (days)	Dependencies
T1	8	
T2	15	
T3	15	T1
T4	10	
T5	10	T2, T4
T6	5	T1, T2
T7	20	T1
T8	25	T4
T9	15	T3, T6
T10	15	T5, T7
T11	7	T9
T12	10	T11

Si imposta la data di inizio del progetto.

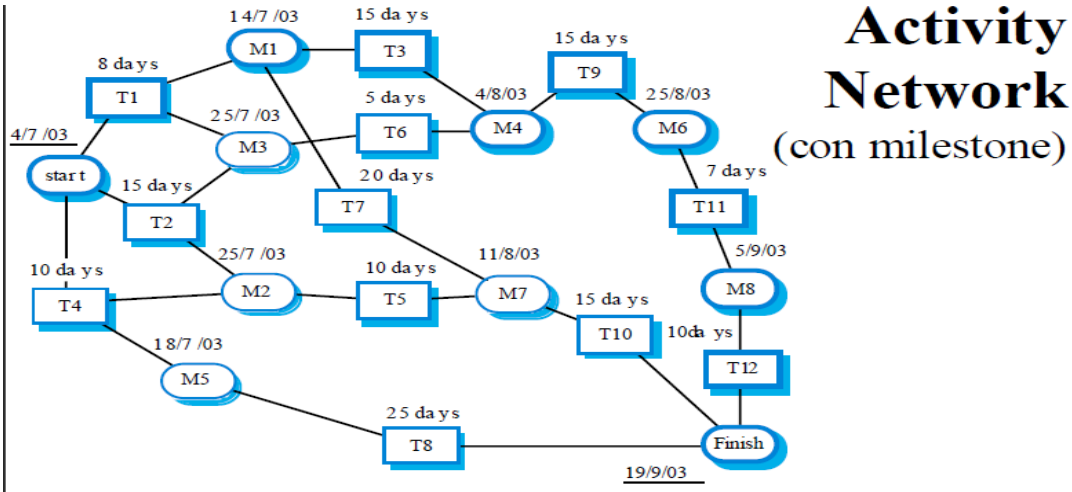
milestone	task
M1	T1
M2	T2, T4
M3	T1, T2
M4	T3,T6
M5	T4
M6	T9
M7	T5, T7
M8	T11

Una dipendenza tra task indica che un task non può iniziare prima che un insieme di task sia stato ultimato (Es.: T5 può iniziare solo dopo T2, T4).

Activity network (senza milestone)



- **Nodi:** task
- **Archi:** dipendenze tra task
- **Cammino critico:** T1, T3, T9, T11, T12: 55 giorni lavorativi (11 settimane)



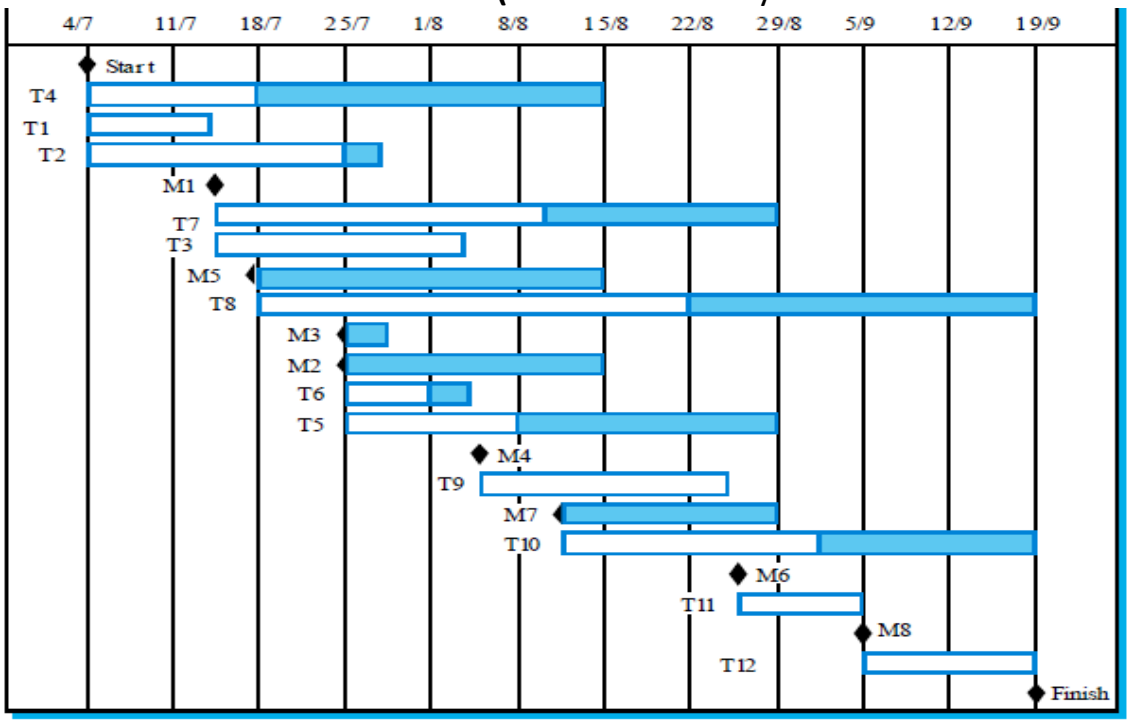
- **Nodi:** task e milestone
- **Archi:** dipendenze tra task, e tra task e milestone
- **Cammino critico:** T1, M1, T3, M4, T9, M6, T11, M8, T12: 55 giorni lavorativi (11 settimane)

Cammino critico

È il cammino più lungo in termini di tempo, dal nodo ‘‘inizio’’ al nodo ‘‘fine’’. Indica il tempo per finire l’intero progetto. Dei ritardi di task nel cammino critico determinano ritardo nel completamento del progetto mentre dei ritardi di task fuori dal cammino critico non determinano un ritardo finale, a meno che tali ritardi non determinino un nuovo cammino critico.

Il **bar chart** indica la durata prevista di un task, e il suo ritardo consentito, senza modificare il cammino critico.

BAR CHART: ACTIVITY TIMELINE (riferito all’activity network con milestone qui sopra**)**



Stima del costo del progetto

Costo del lavoro =Costi fissi dell'azienda + Costo del materiale + Costo dei servizi + Costo dell'hardware + Costo del software (licenze) + ecc

Guadagno dell'azienda informatica

Rischi

Il rischio è una circostanza avversa che si preferirebbe che non accadesse.

–Il rischio ha due caratteristiche:

1)Incertezza: il rischio ha una probabilità di verificarsi

2)Perdita: se il rischio si verifica, comporta delle conseguenze indesiderate

I rischi influenzano negativamente la pianificazione e la qualità del sistema

I rischi si possono classificare

1)in base alla loro fonte (causa)

2)in base a ciò che colpiscono (effetto)

Gestione dei rischi

È il processo che mi permette di identificare i rischi, valutarne l'impatto, definizione di strategie preventive o reattive.

Il processo della gestione dei rischi si divide in quattro fasi:

1)Identificazione rischi: identificare i rischi potenziali e la loro natura (classificazione)

2)Analisi dei rischi: determinare la probabilità di ogni rischio e la gravità delle sue conseguenze

3)Pianificazione dei rischi: piani per affrontare i rischi

4)Monitoring dei rischi: si controlla se la probabilità o la gravità degli effetti di ogni rischio sono cambiate

Classificazione dei rischi in base alla causa

Tipi di rischio (in base alla fonte):

1)Rischi tecnologici: derivano dall'hardware o dal software utilizzati

2)Rischi riguardanti il personale: derivano dalle persone del team

3)Rischi organizzativi: derivano dall'organizzazione aziendale

4)Rischi strumentali: derivano dagli strumenti CASE

5)Rischi dei requisiti: derivano dal cambiamento dei requisiti

6)Rischi di stima: derivano dalle valutazioni relative a tempi, costi, risorse, o caratteristiche del sistema.

Classificazione dei rischi in base all'effetto

1)Rischi di progetto: colpiscono la pianificazione (tempi e risorse)

2)Rischi di prodotto (rischi tecnici): colpiscono la qualità del sistema

3)Rischi di business: colpiscono il committente/cliente o lo sviluppatore sul piano economico

Analisi dei rischi

L'analisi dei rischi è il processo che stima la probabilità del rischio e stima la gravità dei suoi effetti. La valutazione si basa su:

1)La quantità di informazioni sul progetto, sul processo, sul team, sull'organizzazione,

2)Giudizio ed esperienza del manager

Probabilità

–Molto bassa (<10%)

–Bassa (10-25%)

–Moderata (25-50%)

–Alta (50-75%)

–Molto alta (75%)

Effetti possono essere insignificanti, tollerabili, gravi, catastrofici

Pianificazione dei rischi

La pianificazione dei rischi è quel processo che serve per sviluppare strategie per limitare i rischi:

1)Strategie preventive: piani per ridurre la probabilità che il rischio si verifichi

2)Strategie reattive: piani per ridurre gli effetti nel caso il rischio si verifichi

Monitoring dei rischi

Si verifica per ogni rischio identificato, se la sua probabilità può crescere o decrescere, e se la gravità dei suoi effetti può cambiare. Alcuni fattori (indicatori) permettono di monitorare i rischi.

Monitoraggio e revisione del progetto

La gestione del progetto si basa sulle informazioni:

1)Inizialmente le informazioni sono ridotte

- All'inizio si fa una pianificazione, una stima dei costi, ed una gestione dei rischi in modo preliminare

2)Durante il progetto si raccolgono maggiori informazioni, si controlla il rispetto delle milestone, e si possono verificare problemi **(ovvero si fa il monitoraggio)**.

3)Per questo, periodicamente si aggiornano pianificazione, stima dei costi e gestione dei rischi (**revisione**).

- Per i sistemi customizzati, gli aggiornamenti su tempi e costi devono essere concordati con il committente.

Piano di progetto

È un **documento** che raccoglie le informazioni sulle attività di gestione di un progetto:

1)Introduzione: obbiettivi del progetto

2)Organizzazione: persone nel team e ruolo di ciascuno

3)Pianificazione

- **Divisione del lavoro**: divisione del progetto in task, dipendenze, definizione di milestone e deliverable
- **Risorse**: assegnamento delle risorse (persone, hardware, software, budget, ...) ad ogni task; acquisti di hw e sw, personale assunto.
- **Tempistica (Schedule)**: tempi per ogni task, per ogni milestone, per la fine del progetto; tempi per la ricerca di nuove risorse.

4)Gestione dei rischi: possibili rischi, probabilità, effetti, strategie

5)Rapporti sull'avanzamento del progetto.

Viene scritto all'inizio del progetto e viene aggiornato durante il progetto.

MODELLI DI PROCESSO prescrittivi

Attributi del processo software

1)Visibilità: capacità di stabilire ("vedere") lo stato di avanzamento del processo
–in quale fase ci troviamo?
–quante fasi mancano alla fine del processo? •Produzione di documentazione con i risultati di ogni fase

2)Affidabilità: i difetti devono essere scoperti durante il processo software, anziché durante l'uso del prodotto.

3)Robustezza: il processo deve essere in grado di continuare nonostante si presentino problemi inaspettati. –In particolare, il cambiamento dei requisiti

4)Rapidità: tempo necessario per consegnare il sistema

Non è possibile ottimizzare tutti gli attributi poiché ad esempio, rapidità e visibilità sono in contrasto: la produzione di documenti con i risultati di ogni fase rallenta il processo. Gli attributi dipendono dal modello di processo utilizzato.

Modello di processo: È l'organizzazione delle fasi del processo software
–Organizzazione di riferimento: si può applicare alla lettera oppure ispirare altre soluzioni.

PRIMO MODELLO: Modello a cascata (waterfall) (Royce, 1970)
Fasi sequenziali (a cascata): Prima di poter passare alla fase successiva, la fase corrente deve essere stata completata.
Tale modello è un processo black-box per il committente.
Il committente tramite questo modello è coinvolto solo nella fase di specifica e può visionare il sistema solo alla consegna quindi dopo le fasi di progettazione, implementazione e collaudo.

Modello a cascata: attributi
1)Visibilità alta:
- –Ogni fase produce della documentazione che ne descrive i risultati.
- –Le fasi sono ben distinte (senza sovrapposizione)
  - •Prima forma storica di modello di processo: deriva direttamente dal processo dell'ingegneria tradizionale.

2)Affidabilità bassa:
- –Verifica: si collauda il sistema in blocco
- •È più difficile trovare difetti in una grande quantità di codice
- –Validazione: il committente può controllare il soddisfacimento dei requisiti solo alla consegna.
- •È tardi per accorgersi di difetti nei requisiti: bisogna correggere specifica, progettazione, implementazione.

3)Robustezza bassa: i requisiti non devono cambiare
- –difficoltà ad effettuare cambiamenti ai requisiti dopo la specifica
  - •il sistema può essere già interamente progettato, e magari già implementato

4)Rapidità bassa: il committente può disporre del sistema solo alla consegna, dopo una lunghissima attesa dovuta a
1)Lunga fase di specifica in cui i requisiti devono essere tutti definiti in dettaglio
2)Lunga fase di progettazione che deve essere dettagliata e completa
3)Lunga fase di implementazione in cui bisogna implementare l'intero sistema
4)Lunga fase di collaudo in cui bisogna collaudare tutto il sistema

Alcuni sviluppatori potrebbero andare in "stato bloccante", cioè attendere la fine dell'attività di un collega
1)I progettisti devono aspettare la fine della specifica
2)I programmatori devono aspettare la fine della progettazione
3)I collaudatori devono aspettare la fine dell'implementazione

Commenti sul modello a cascata

I requisiti sono definiti una volta per tutte, all'inizio. Quindi devono essere:

1)ben compresi;
2)definiti in modo completo e dettagliato;
3)stabili (''congelati'').

Nella realtà, raramente è così

1)difficoltà da parte del committente nell'esprimere i requisiti
2)il committente ha un'idea generale dei requisiti del sistema
3)capita che il committente cambi idea dopo la specifica.

Nella realtà, le fasi non sono così sequenziali

1)Durante la progettazione si scoprono problemi nei requisiti, e quindi si ritorna alla specifica
2)Durante l'implementazione si scoprono problemi nella progettazione, e quindi si ritorna alla fase di progettazione
3)Se mi accorgo di errori in fasi precedenti posso tornare indietro

Prototipo

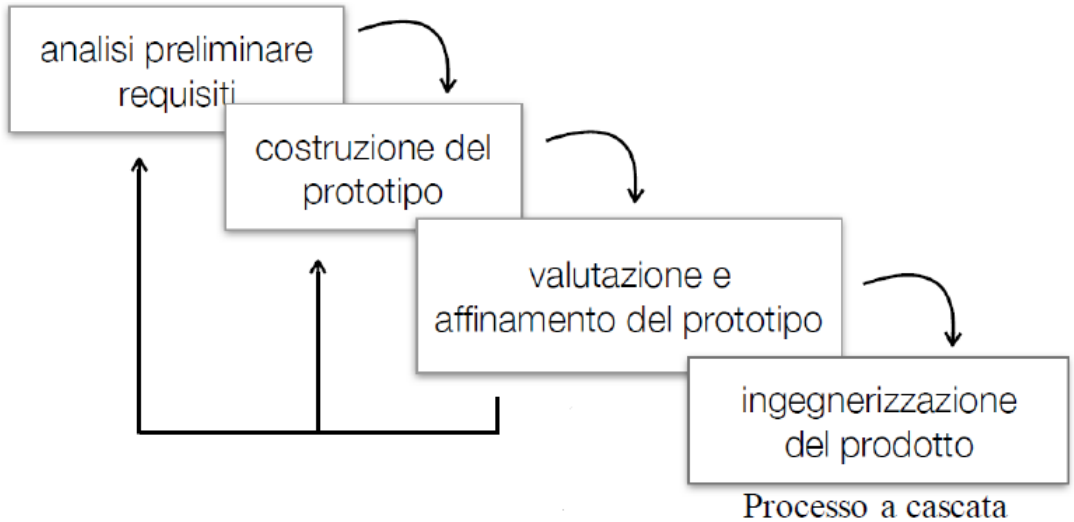
Il prototipo è una versione preliminare del sistema che realizza un sottoinsieme dei requisiti

Prototipi "usa e getta":

Lo scopo del prototipo è comprendere un insieme di requisiti poco chiari; il prototipo realizza tale insieme di requisiti. Il prototipo non ha la qualità del prodotto finale (si trascurano i requisiti non funzionali). Il prototipo viene mostrato al committente e valutato. Il prototipo non evolve nel sistema finale, ma viene "gettato via".

Il prototipo è utile se è combinato con il **modello a cascata**:

1)Nella fase di specifica (validazione dei requisiti) si producono e si valutano uno o più prototipi
2)Si chiariscono i requisiti
3)Si sviluppa il vero sistema con il modello a cascata

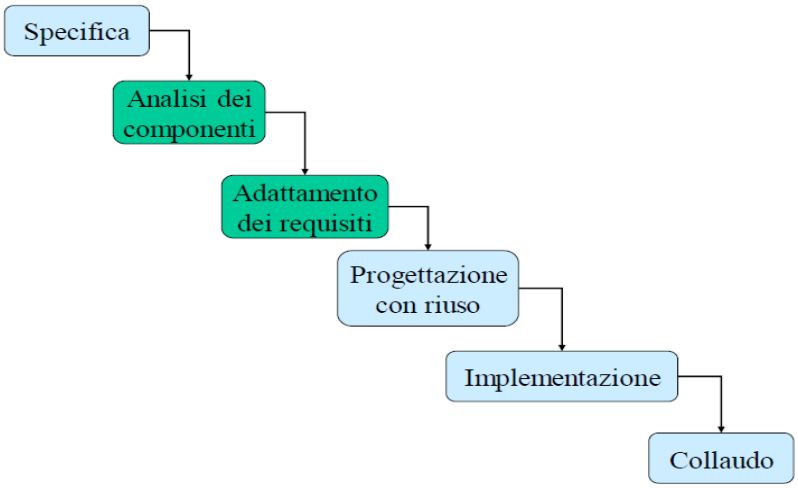


Processo a cascata

Altri usi del prototipo “usa e getta”:
1)Sperimentare soluzioni software: Architetture e Linguaggi di programmazione
2)Back-to-back testing: il sistema da consegnare e un prototipo alternativo sono sottoposti agli stessi test. A questi test i due sistemi devono dare gli stessi risultati, altrimenti c’è un errore.
3)Training degli utenti in attesa del vero sistema

Un prototipo “usa e getta” ha un costo. Si può recuperare il costo attraverso un risparmio durante lo sviluppo del sistema, grazie ai requisiti più chiari.

Sviluppo basato sul riuso (Component Based Software Engineering (CBSE))
Finora si è fatta l’assunzione che tutti i componenti del sistema siano sviluppati ad-hoc.
CBSE si basa sul riuso di componenti **esistenti**
1)realizzati per altri sistemi
2)componenti commerciali COTS (commercial off the shelf).
Componenti sviluppati ad-hoc possono essere integrati con i componenti esistenti.
L’uso di questo approccio è sempre più frequente a causa dell’affermarsi di componenti standard (Esempio: DBMS).
Sviluppo basato sul riuso



1)Specifica
2)Analisi dei componenti: si valutano quali dei componenti esistenti possono essere riutilizzati.
3)Adattamento dei requisiti: si modificano i requisiti in modo che possano essere realizzati attraverso i componenti esistenti.
4)Progettazione con riuso: definizione dell’architettura del sistema composta da componenti esistenti e componenti ad-hoc da implementare.
5)Implementazione: codifica dei componenti ad-hoc ed integrazione con quelli esistenti.
6)Collaudo

Sviluppo basato sul riuso: attributi
Confronto con modello a cascata:
1)Stessa **visibilità** (alta)
2)**Affidabilità** migliore (media): i componenti pre-esistenti sono già stati collaudati in precedenza (*verifica*)
3)**Robustezza** migliore (media):

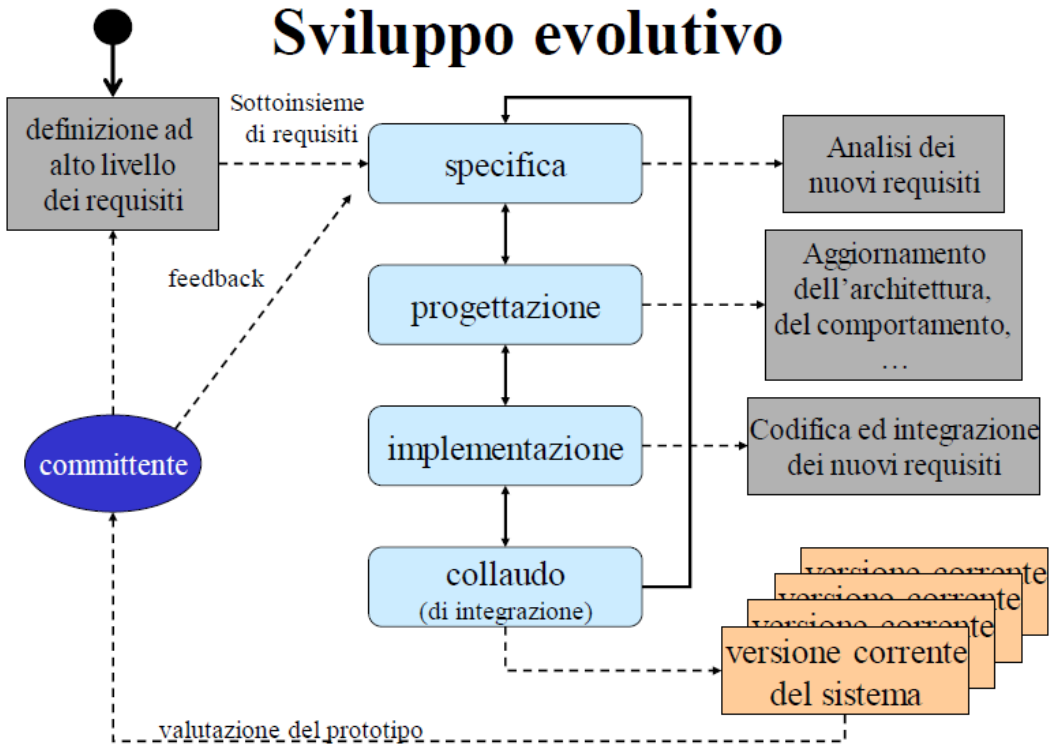
- necessità di compromessi sui requisiti: i componenti esistenti potrebbero non essere perfettamente adatti a soddisfare i requisiti
- Riduzione del costo: un componente esistente costa meno che un componente sviluppato ad-hoc
- Riduzione dei rischi: limitati ai nuovi componenti

4)**Rapidità** migliore (media): una sola consegna, ma più veloce, dato che alcuni componenti sono già pronti all'inizio del processo

SECONDO MODELLO: Modelli iterativi
Tale processo è white box per il committente
Con questo modello si prevede la ripetizione di alcune fasi del processo e si prevede la partecipazione del committente durante tutto il processo per verificare il soddisfacimento dei requisiti.

Il sistema è un **prototipo evolutivo**: il prototipo evolve nel sistema finale (non viene gettato via) aggiungendo gradualmente i requisiti mancanti.

Il costo delle modifiche con il modello a cascata ha un andamento esponenziale mentre con il modello iterativo a un andamento logaritmico.



Sviluppo evolutivo: fasi

1)Definizione ad alto livello dei requisiti

2)Si seleziona un sottoinsieme dei requisiti

3)Si aggiorna il prototipo corrente facendo la specifica, la progettazione, l'implementazione e il collaudo dei nuovi requisiti. In ogni fase, è possibile ritornare a quella precedente per risolvere problemi.

4)Si genera una release del sistema che realizza i requisiti considerati finora (prototipo). Si mostra il prototipo al committente che fornisce un feedback.

5)Se necessario, si corregge il prototipo in base al feedback (aggiornando specifica, progettazione ed implementazione).

6)Se ci sono altri requisiti da trattare, si ritorna al passo 2

Il sistema è realizzato tramite un **prototipo evolutivo**:

1)Dopo il primo ciclo, il prototipo realizza il primo sottoinsieme di requisiti.

2)Ad ogni ciclo successivo, il prototipo mantiene i requisiti precedenti e viene esteso con altri requisiti.

3)Si itera il processo fino a quando tutti i requisiti sono stati considerati.

I requisiti da trattare vengono decisi all'inizio di ogni ciclo (passo 2).

Sviluppo evolutivo: requisiti

Si incomincia con i requisiti funzionali più chiari e con maggiore priorità. I requisiti non funzionali si trattano successivamente (efficienza, affidabilità, …).

Il committente valuta ogni versione del prodotto fornendo un **Feedback** in cui indica:

1)Indica se i requisiti implementati non sono soddisfatti

2)Chiarisce i requisiti poco chiari (ancora da implementare)

3)Introduce nuovi requisiti

Sviluppo evolutivo: attributi

1)Visibilità bassa:

La documentazione non viene sempre aggiornata sarebbe troppo costoso se le versioni intermedie fossero numerose e non si sa quante versioni mancano alla fine del processo.

2)Affidabilità alta:

–Verifica: il collaudo riguarda un sottoinsieme di requisiti

- È più facile trovare difetti se il codice è limitato poiché i bachi sono scoperti anticipatamente

–Validazione: controllo del soddisfacimento dei requisiti per ogni versione, grazie al coinvolgimento del committente

- Scoperta anticipata di equivoci tra committente e sviluppatori
- Scoperta anticipata di requisiti incompleti o mancanti

3)Robustezza media:

Il processo supporta il cambiamento dei requisiti;

–Specifica e progettazione non devono essere definite in modo dettagliato all'inizio del processo. Una parte dei dettagli si definiscono ad ogni ciclo

–Il committente può cambiare idea sui requisiti durante il processo. È possibile che il requisito da cambiare non sia stato ancora analizzato, progettato e implementato.

Il codice rischia di diventare poco strutturato

–I continui cambiamenti all'implementazione possono corrompere gradualmente la struttura del codice: i cambiamenti diventano sempre più faticosi.

4)Rapidità media:

È presente una disponibilità anticipata di un prototipo funzionante (non bisogna aspettare la fine del processo)

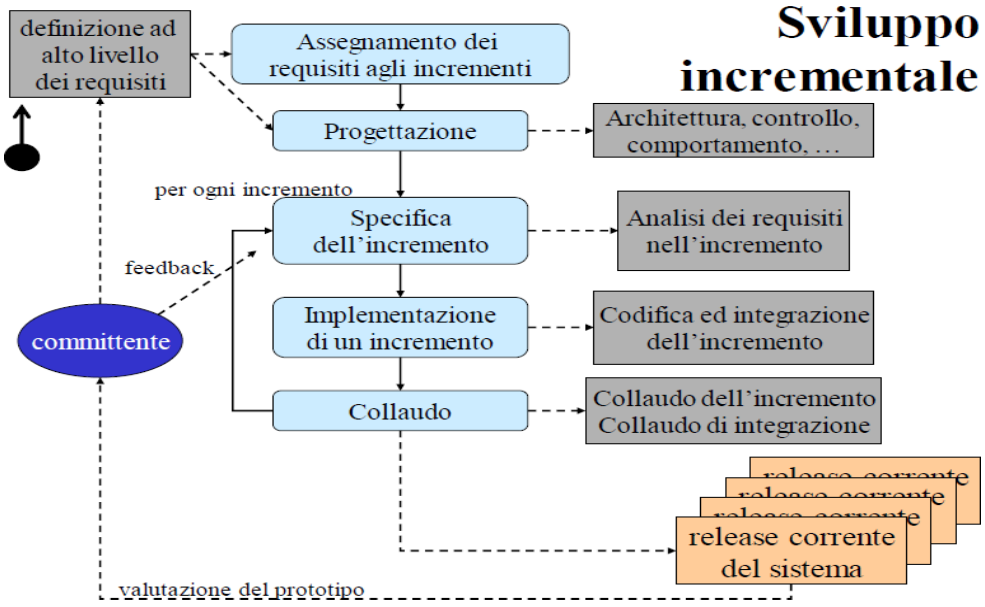
- Training anticipato all'uso (parziale) del sistema
- Scoperta anticipata di difficoltà d'uso da parte dell'utente

–La consegna di una versione richiede comunque di aspettare il tempo necessario per l'analisi, la progettazione, l'implementazione e il collaudo dei relativi requisiti.

- Il committente non deve aspettare troppo a lungo le release del sistema
- Il committente deve essere disponibile a fornire un feedback
- Non tutti i committenti sono disponibili a partecipare al processo

Sviluppo evolutivo: non è prevedibile il numero di versioni intermedie necessarie per arrivare alla versione completa del sistema (visibilità bassa)

Sviluppo incrementale: mantiene i vantaggi dello sviluppo evolutivo (sviluppo del sistema tramite un prototipo evolutivo) però, ha una maggiore visibilità (definizione a priori del numero di versioni del prototipo per arrivare al sistema finale)



Sviluppo incrementale: fasi

•**Fasi iniziali**:

1)Specifica ad alto livello

2)Si stabilisce quale sarà il numero di **incrementi** (versioni) del sistema durante lo sviluppo

- •Ogni incremento realizza un sottoinsieme dei requisiti

3)Si stabilisce l'ordine degli incrementi

- •Ogni incremento ha una priorità in base all'importanza del requisito

–Viene progettato il sistema

•**Fasi cicliche** (iterazione per ogni incremento, in ordine di priorità):

1)Specifica in dettaglio dei requisiti dell'incremento

2)Implementazione dell'incremento: codice ed integrazione

3)Collaudo: dell'incremento e di integrazione

4)Generazione della release corrente del sistema

- •Il committente la esamina e fornisce un feedback
- •Se necessario, si fanno correzioni aggiornando specifica ed implementazione

Sviluppo incrementale: attributi

1)Visibilità alta: Il processo è meno soggetto all'incertezza; all'inizio del processo si definiscono:

–L'architettura del sistema

–Il numero di incrementi (versioni) e i requisiti realizzati in ogni incremento

- •Si conosce sempre il numero di versioni che servono per arrivare alla fine
- •È più facile pianificare la documentazione sapendo il numero di versioni

–L'ordine degli incrementi

2)Affidabilità e **Rapidità**: come per sviluppo evolutivo

3)Robustezza media: supporta il cambiamento dei requisiti

–I requisiti non devono essere definiti in modo dettagliato all'inizio del processo. Un sottoinsieme dei requisiti è analizzato ad ogni ciclo.

–Il committente può cambiare idea sui requisiti durante il processo, purché non comporti un cambiamento dell'architettura (già definita all'inizio)

–Conoscendo in anticipo l'architettura e il numero di incrementi, il codice può essere strutturato più ordinatamente.

Processo unificato (Unified Process, UP): Si suddivide in 4 fasi:

1)Avvio (inception): fa ''decollare'' il progetto (studio di fattibilità, requisiti preliminari, rischi preliminari, ...)

2)Elaborazione: genera una ''baseline'', cioè un primo prototipo (evolutivo) del sistema

3)Costruzione: si passa dalla baseline al sistema finale

4)Transizione: prepara tutto ciò che è necessario per la consegna del sistema presso il committente (adattare sistema a piattaforma hw/sw, manuale utente, sito web, ...)

UP: workflow

Ogni fase richiede l'esecuzione di questi **workflow**: requisiti, analisi, progettazione, implementazione, test.

La quantità di lavoro (tempo) dedicato ad ogni workflow varia a seconda della fase.

Una fase può essere eseguita più volte (iterazioni).

Alla fine di ogni fase (o ripetizione di una fase) si ottengono:

–Milestone (per valutare l'avanzamento del lavoro)

–Una versione (prototipo) eseguibile

UP: caratteristiche

UP è contemporaneamente

1)Sequenziale: 4 fasi in sequenza

2)Iterativo: Ogni fase può essere ripetuta (iterazione) oppure ogni fase o iterazione di fase ripete gli stessi workflow

3)Incrementale: ogni fase produce un prototipo evolutivo

- UP considera la gestione dei rischi
- **Attributi**: gli stessi dello sviluppo evolutivo

UP considera l'uso di UML (stessi autori):

1)Avvio: diagrammi per dedurre i requisiti (diag. classi del dominio, diag. casi d'uso preliminare)

2)Elaborazione: diagrammi per

- l'analisi di requisiti (diag. casi d'uso definitivo, diagramma degli stati)
- la strutturazione del sistema (diag. dei componenti, diag. di deployment, diag. classi dell'architettura, diag. di package)

3)Costruzione: diagrammi per modellare il comportamento del sistema (diag. di sequenza, diag. di attività).

Modelli a spirale

Le attività sono distribuite in vari settori del piano. Il processo avviene attraversando varie volte tali settori secondo un percorso a spirale.

I modelli a spirale considerano anche le **attività di gestione e i prototipi**.

- Versione I: prevede il rilascio del sistema alla fine dell'intero processo
- Versione II: prevede lo sviluppo del sistema tramite un prototipo evolutivo (varie release intermedie)

Modello a spirale (versione I, Boehm, 1988)

Include le attività di gestione (manager). Ogni **loop** rappresenta un task del processo.

–Ogni loop è diviso in 4 **settori**:

- Determinazione di obbiettivi e alternative
- Valutazione delle alternative e dei rischi
- Sviluppo e convalida
- Pianificazione

Il processo inizia con la pianificazione della prima fase.

Spirale I: settori di un loop

- **Determinazione** di
  - –**obbiettivi** specifici del task corrente (es.: deduzione dei requisiti, implementazione di una funzione, miglioramento performance, …)
  - –Possibili **soluzioni** per il task in corso
    - Le soluzioni sono tra loro alternative
- **Valutazione rischi e alternative**
  - –Gestione dei rischi
  - –Costruzione di prototipi per valutare le varie soluzioni alternative
- **Sviluppo e convalida**
  - –Sviluppo di modelli, simulazioni, test per valutare i vari prototipi
  - –Scelta della soluzione in base alla valutazione del prototipo
  - –Sviluppo e verifica dell'attività in base alla soluzione scelta
- **Pianificazione**
  - –Il progetto viene revisionato
  - –Pianificazione della fase successiva

Modello a spirale (II versione, Boehm, 1998)

- Processo ciclico: una release ad ogni loop
- Ogni settore di un loop è una fase del processo
- Include le attività di gestione del progetto
- Fasi ben distinte

Spirale I e II: attributi

Spirale I: confronto con modello a cascata

–**Visibilità** (alta) e **Rapidità** (bassa) sono le stesse

–**L'Affidabilità** è migliore (media) grazie all'uso di prototipi ''usa e getta'' per la validazione dei requisiti.

–La **Robustezza** è migliore (media) grazie all'uso di prototipi, modelli, simulazioni, analisi dei rischi, che dovrebbero limitare il verificarsi di problemi. Comunque, i requisiti devono essere stabili.

- **Spirale II**: confronto con sviluppo evolutivo

–Stessi attributi.

MODELLI DI PROCESSO ‘‘agili’’

Sviluppo rapido del software

Per rispondere alla pressione concorrenziale, oggi il software deve essere consegnato in modo rapido.

→Il modello a cascata è poco adatto alla rapidità poiché:

- I requisiti non sono tipicamente stabili,
- Fasi lunghe,
- Consegna solo quando il sistema è completo.

→Sviluppo evolutivo ed incrementale sono migliori per la rapidità

- Consegna anticipata di una parte del sistema (prototipo evolutivo).

Per team composti da poche persone, e per progetti limitati, seguire un modello di processo “prescrittivo” può generare overhead.

–I modelli precedenti sono troppo vincolanti per un piccolo team

–Rispettare le fasi del processo può rubare tempo utile all’implementazione e posticipare la consegna.

–In un gruppo piccolo, non ci sono ruoli precisi (ognuno fa un po’ di tutto)

Approccio “Agile”

Negli anni ’90 iniziano a nascere aziende informatiche di piccole dimensioni (prima erano solo grandi e poche). Ai piccoli team servono dei modelli di processo più “agili”. Nell’approccio **“agile”** si è:

–**Pronto al cambiamento**: è scontato che i requisiti cambieranno.

–**Flessibile**: le persone non si adattano ad un modello di processo “prescrittivo”, ma è il processo che si adatta alle esigenze delle persone.

Lo scopo è la consegna **rapida** del prodotto. Riguarda progetti limitati, realizzati da team di sviluppo composti da poche persone. Non riguarda progetti industriali.

Il manifesto agile (da agilemanifesto.org)

«Stiamo scoprendo modi migliori di creare software, sviluppandolo e aiutando gli altri a fare lo stesso. Grazie a questa attività siamo arrivati a considerare importanti:

–Gli individui e le interazioni più che i processi e gli strumenti

–Il software funzionante più che la documentazione esaustiva

–La collaborazione col cliente più che la negoziazione dei contratti

–Rispondere al cambiamento più che seguire un piano»

«Ovvero, fermo restando il valore delle voci a destra, consideriamo più importanti le voci a sinistra.»

I principi del manifesto agile

•«La nostra massima priorità è soddisfare il cliente rilasciando software di valore, fin da subito e in maniera continua.

•Accogliamo i cambiamenti nei requisiti, anche a stadi avanzati dello sviluppo. I processi agili sfruttano il cambiamento a favore del vantaggio competitivo del cliente.

•Consegniamo frequentemente software funzionante, con cadenza variabile da un paio di settimane a un paio di mesi, preferendo i periodi brevi.

•Committenti e sviluppatori devono lavorare insieme quotidianamente per tutta la durata del progetto.

•Fondiamo i progetti su individui motivati. Diamo loro l'ambiente e il supporto di cui hanno bisogno e confidiamo nella loro capacità di portare il lavoro a termine.

•Una conversazione faccia a faccia è il modo più efficiente e più efficace per comunicare con il team ed all'interno del team.

•Il software funzionante è il principale metro di misura di progresso.

•I processi agili promuovono uno sviluppo sostenibile. Gli sponsor, gli sviluppatori e gli utenti dovrebbero essere in grado di mantenere indefinitamente un ritmo costante.

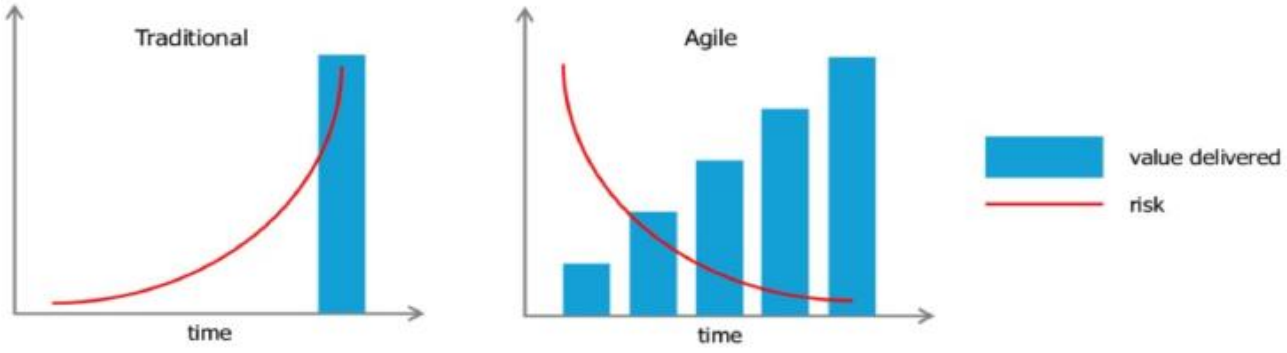
•La continua attenzione all'eccellenza tecnica e alla buona progettazione esaltano l'agilità.

•La semplicità - l'arte di massimizzare la quantità di lavoro non svolto - è essenziale.

•Le architetture, i requisiti e la progettazione migliori emergono da team che si auto-organizzano.

•A intervalli regolari il team riflette su come diventare più efficace, dopodiché regola e adatta il proprio comportamento di conseguenza.»

Modello tradizionale (o a cascata) vs modello agile



eXtreme Programming (XP)

È il più noto modello di processo tra quelli ‘‘agili’’. XP è una forma “estrema” di **sviluppo evolutivo**:

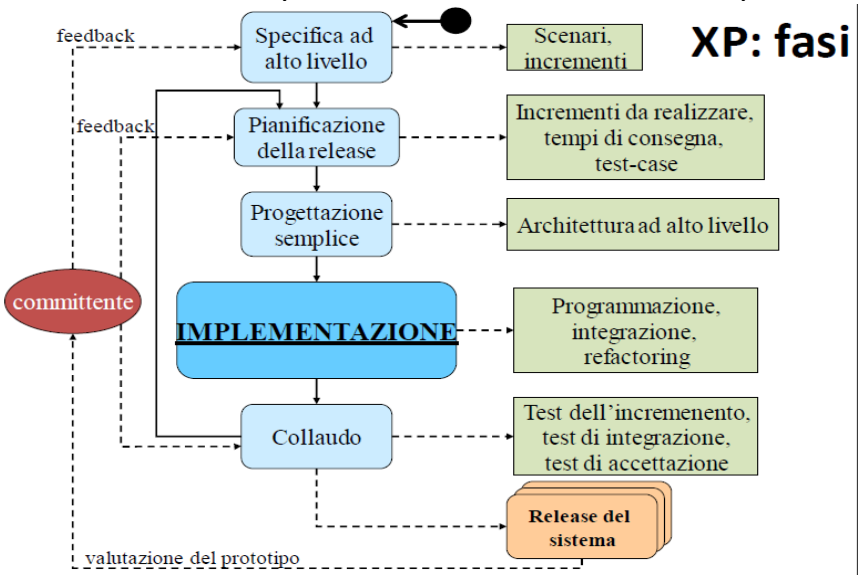
–La **rapidità** è l’obbiettivo fondamentale

- •I tempi di consegna sono ristretti
- •L’**implementazione** (programming) è la fase su cui ci si concentra. Le altre fasi si eseguono solo ad alto livello, per guadagnare tempo
- •Servono accorgimenti che garantiscano la **qualità** del prodotto nonostante la rapidità dello sviluppo.

–Numerose release del sistema

–Prevede il cambiamento dei requisiti

–Il committente è particolarmente coinvolto nel processo



XP: fasi

1)Specifica a (molto) alto livello:

–I requisiti sono definiti soltanto ad alto livello, sottoforma di scenari non strutturati

–Gli scenari sono ordinati per priorità

–Ogni scenario viene suddiviso in incrementi

–Una specifica dettagliata richiederebbe più tempo riducendo così la rapidità del processo

2)Pianificazione:

–Si stabiliscono assieme al committente

- •Quali incrementi saranno realizzati dalla prossima release
  - –Di solito una release realizza uno o pochi incrementi
- •I tempi di consegna della prossima release
- •I test da eseguire in fase di collaudo

3)Progettazione semplice:

–Si definisce o si aggiorna l'architettura a (molto) alto livello

- La progettazione definisce solo ciò che è strettamente necessario

–I dettagli saranno stabiliti mentre si implementa

–Una progettazione dettagliata richiederebbe più tempo riducendo così la rapidità del processo

4)Implementazione:

–**Programmazione a coppie**: i programmatori lavorano a coppie ("2 menti sono meglio di una")

- Ogni linea di codice è scritta e visionata da due persone: aumenta la probabilità di "vedere" un difetto (prima modalità di collaudo)
- Il driver scrive il codice, il *navigatore* può controllarlo e fare **refactoring**
  - –Codice più pulito, uniforme e capibile: stile di indentazione, posizione delle parentesi graffe, stile dei nomi, posizione commenti, ...
  - –Codice più efficiente: evitare ridondanze, variabili e cicli inutili, ...
  - –Non si altera il comportamento, ma diventa più mantenibile
- Condivisione della conoscenza del codice
  - –Le coppie cambiano dinamicamente a seconda degli incrementi.
  - –Quando uno sviluppatore cambia compagno può spiegargli la parte di codice che aveva scritto.
  - –Gradualmente, cambiando le coppie varie volte, ogni parte di codice è conosciuta da un numero maggiore di sviluppatori.
  - –Alla fine, ogni sviluppatore conosce tutto il codice.

–**Sviluppo con test iniziale**: i test-case sono definiti durante la fase di pianificazione, prima dell'implementazione dell'incremento

- Il programmatore scrive il codice in modo che il test venga superato
- Test-Driven Development (TDD)

–**Possesso collettivo**: un programmatore può modificare qualunque parte del codice

5)Collaudo: I test-case sono scelti prima dell'implementazione. I tre **tipi** di testing ad ogni incremento sono:

- **Testing dell'incremento (unit test)**: fatto sull'incremento
  - –Se l'incremento supera il test, viene integrato
  - –Test-case stabiliti dagli sviluppatori
- **Test di integrazione**: tutti i test di unità precedenti sono ripetuti quando si integra un incremento
- **Test di accettazione**: il committente controlla il soddisfacimento dei requisiti
  - –test case stabiliti da sviluppatori e committente

Si usano strumenti CASE per il testing che riducono i tempi del collaudo, quindi migliora la rapidità.

XP: ruolo del committente: Il committente (o un rappresentante) è molto più coinvolto nel processo, rispetto ai modelli precedenti:

- Assegna le priorità ai requisiti
- Decide cosa deve contenere ogni release
- Decide le scadenze per le release
- Partecipa alla definizione dei requisiti (scenari)
- Partecipa alla definizione dei test-case
- Partecipa all'esecuzione dei test-case
- Si trova nella stessa sede degli sviluppatori

XP: commenti

Committente on-site: presenza a tempo pieno del committente (o di un suo rappresentante) nel team di sviluppo:

- Il committente deve essere una singola entità
- Deve essere costantemente a disposizione del team di sviluppo

I programmatori lavorano su ogni parte del sistema (**possesso collettivo**):

- devono avere varie conoscenze
- devono essere flessibili
- non ci sono ruoli prestabiliti
- devono lavorare nella stessa sede
- chiunque può modificare qualunque parte del sistema
- chiunque trova un problema lo risolve

XP: attributi

1)Visibilità bassa:

- Non c'è la documentazione nella sua forma classica.
  - Documentazione in XP = scenari, codice, test-case
- Non si conosce il numero di versioni per arrivare alla fine
  - I requisiti di ogni versione sono scelti di volta in volta

2)Rapidità alta: è ottenuta attraverso

- Specifica a (molto) alto livello
- Progettazione a (molto) alto livello
- Uso di strumenti CASE durante il collaudo
- Release molto frequenti (ogni 1-2 settimane, *time boxed*)
  - ogni release realizza pochi incrementi quindi lo sviluppo richiede poco tempo

3)Affidabilità alta: La qualità del prodotto è mantenuta nonostante la rapidità alta.

–Verifica

•Sviluppo con test iniziale (Test-Driven Development)

•Release frequenti: tante release, tanti collaudi

–Aumenta la probabilità di trovare un difetto

•Ogni collaudo riguarda piccoli incrementi

–È più facile trovare un difetto se il codice è limitato e scritto recentemente

–Validazione: coinvolgimento costante del committente:

•Il committente valuta frequentemente il prodotto fornendo feedback sul soddisfacimento dei requisiti

4)Robustezza alta

–Supporta il cambiamento dei requisiti

→Prototipo evolutivo

→Committente on site: partecipa alla stesura di scenari ed incrementi

–Il codice rimane ben strutturato e ordinato

→Programmazione a coppie con refactoring

XP: refactoring

•Il codice è soggetto a frequenti modifiche.

–Fase 1: modifica per aggiornare il codice

–Fase 2: testing

–Fase 3: refactoring per mantenerlo ‘‘pulito’’

–Fase 4: rifare testing (test di regressione)

•Il refactoring non modifica il comportamento del codice, ma ne migliora la struttura interna.

•Eliminare codice duplicato (tramite funzioni/metodi)

•Ridurre la lunghezza di una funzione/metodo distribuendo il codice in varie funzioni/metodi

•Dare nomi significativi a variabili, funzioni, classi, metodi, attributi, per far capire il loro scopo.

•Fondere classi simili in un’unica classe.

•Uniformare l’indentazione (spazi, tab, posizione { }) e la posizione dei commenti.

•Codice semplice e uniforme → più leggibile

Esempio di refactoring

```
if (isSpecialDeal()) {
        total = price * 0.95;
        send();
}
else {
        total = price * 0.98;
        send();
}
```

dopo il refactoring diventa→

```
if (isSpecialDeal())
        total = price * 0.95;
else
        total = price * 0.98;
send();
```

XP: test driven development (TDD)

Invece di scrivere prima il codice e poi i test-case, si scrivono prima i test-case e poi il codice, in modo da superarli.

1)Scrivi test che fallisce
2)Scrivi codice che supera il test
3)Refactoring + Ripeti test
4)Go To 1.

TDD: esempio

Scrivere un metodo che prende in ingresso due valori interi (a, b) e restituisce un valore intero uguale alla somma di (a, b) se (a, b) sono entrambi positivi altrimenti -1.

TDD: test 1

```
//metodo
public static Object sum (Integer a, Integer b)
{
   return new Object();
}
```

```
//Test
Integer a = new Integer(5);
Integer b = new Integer(2);

Object o = sum(a,b);
if (o instanceof Integer)
   return true;
else
   return false;
```

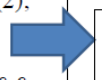
```
//metodo
public static Integer sum (Integer a, Integer b)
{
   return new Integer();
}
```

TDD: test 2

```
//metodo
public static Integer sum (Integer a, Integer b)
{
   return new Integer();
}
```

```
//Test
Integer a = new Integer(5);
Integer b = new Integer(2);

Object o = sum(a,b);
if (o instanceof Integer &&
new Integer(7).equals(o))
   return true;
else
   return false;
```



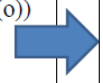
```
//metodo
public static Integer sum (Integer a, Integer b)
{
   return new Integer(a.intValue() + b.intValue());
}
```

TDD: test 3

```
//metodo
public static Integer sum (Integer a, Integer b)
{
   return new Integer(a.intValue() + b.intValue());
}
```

```
//Test
Integer a = new Integer(-5);
Integer b = new Integer(2);

Object o = sum(a,b);
if (o instanceof Integer &&
new Integer(-1).equals(o))
   return true;
else
   return false;
```



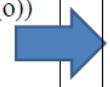
```
//metodo
public static Object sum (Integer a, Integer b)
{  if (a<0)
       return -1;
   else
      return new Integer(a.intValue() + b.intValue());
}
```

TDD: test 4

```
//metodo
public static Object sum (Integer a, Integer b)
{  if (a<0)
       return -1;
   else
      return new Integer(a.intValue() + b.intValue());
}
```

```
//Test
Integer a = new Integer(-5);
Integer b = new Integer(-2);

Object o = sum(a,b);
if (o instanceof Integer &&
new Integer(-1).equals(o))
   return true;
else
   return false;
```



```
//metodo
public static Object sum (Integer a, Integer b)
{  if (a<0 || b<0)
       return -1;
   else
      return new Integer(a.intValue() + b.intValue());
}
```

Scrum

È un altro modello di processo ‘‘agile’’.

Scrum: ruoli e processo

Scrum = ‘‘pacchetto di mischia’’ (rugby) = squadra coesa e compatta

- **Ruoli:**
  1) Product owner = committente
  2) Scrum team = 6-10 sviluppatori
  3) Scrum master = manager del team
- **Fasi del processo:**

Definizione del product backlog = insieme dei requisiti

Da 3 a 8 sprint e ogni sprint contiene micro-waterfall (progettazione, implementazione, collaudo) genera un prototipo evolutivo da mostrare al product owner. Dura 1 mese (time boxed).

Scrum: backlog

- **Product backlog** = elenco di tutti i requisiti e relative priorità
  - –Definito all'inizio del processo dal product owner
  - –Può essere aggiornato dopo ogni sprint (modifica di requisiti e priorità) dallo scrum master
  - –I requisiti con priorità più alta saranno sviluppati prima
- **Sprint backlog**: sottoinsieme del product backlog da sviluppare durante uno specifico sprint

Scrum: meeting

- **Sprint planning meeting** = riunione all'inizio di ogni sprint
  - –Partecipanti: *product owner*, *scrum master*, *scrum team*
    - Aggiornamento del *product backlog* (se necessario)
    - Definizione dello *sprint backlog* dello sprint corrente
    - Definizione dello *sprint goal* (obbiettivo dello sprint corrente)
- **Scrum meeting** (daily scrum) = riunione giornaliera
  - –Partecipanti: scrum master, scrum *team* (15-30 minuti)
    - Ogni sviluppatore descrive cosa ha fatto il giorno prima, gli impedimenti riscontrati, cosa farà nel giorno corrente.
    - Scrum master controlla l'avanzamento del lavoro, aggiorna lo *sprint* backlog (se necessario), aiuta a superare gli impedimenti e ad impostare il lavoro del giorno corrente.
- **Sprint review meeting** = riunione alla fine di ogni sprint
  - –Partecipanti: *product owner*, *scrum master*, *scrum team*
    - Si mostra il prototipo corrente
    - Il product owner indica se lo *sprint goal* è stato raggiunto o meno

Scrum: commenti

- Le riunioni giornaliere promuovono la comunicazione tra gli sviluppatori, la conoscenza collettiva e la responsabilità collettiva del progetto (gli sviluppatori formano uno **scrum**).
- Il coinvolgimento del committente limita i fraintendimenti sui requisiti.
- Scrum è più un modello gestionale che di processo
  - –Non fornisce indicazioni sulle fasi fondamentali (specifica, progettazione, implementazione, collaudo)
  - –Per questo, Scrum può essere integrato con altri modelli (XP@Scrum, UP@Scrum)

Scrum: attributi

- **Visibilità media**:
  - –Controllo giornaliero dell'avanzamento del lavoro
  - –Non si conosce a priori il numero di sprint
- **Affidabilità alta:**
  - –Coinvolgimento costante (mensile) del committente
  - –Il collaudo è effettuato per ogni prototipo
- **Robustezza alta**:
  - –Gli impedimenti sono notificati e risolti giornalmente
  - –Il backlog può essere modificato dopo ogni sprint
- **Rapidità alta:**
  - –Il prototipo viene rilasciato ogni mese

DevOps

Collaborazione tra

- -Il team di sviluppo (**Dev**)
- -Gli amministratori dei sistemi (**Ops**)

Concetti chiave:

-**Dev (Build)**: plan, code, build, test

-**Release**

-**Ops (Operate)**: deploy, operate, monitor

-**Feedback**

Perché DevOps?

- Passaggio da ''Systems of record'' a ''Systems of engagement''
  - –Systems of record: sistemi software ''tradizionali''
    - Registrano (record) grandi quantità di dati e transazioni.
    - Stabili e addifabili: una o due release all'anno.
  - –Systems of engagement: applicazioni mobili, cloud, big data, social media, ...
    - Maggiore frequenza di utilizzo da parte dell'utente (engagement).
    - Richiesta di alte prestazioni.
    - Tempi rapidi di sviluppo e manutenzione (i requisiti cambiano spesso in base alle richieste degli utenti e al mercato).

DevOps: soggetti e scopi

- Soggetti coinvolti:
  - –Esperti del dominio (Dev)
  - –Sviluppatori (Dev)
  - –Amministratori di sistema (Ops)
  - –Addetti alla qualità del software (Ops)
  - –Utenti finali (Ops)
- Scopi:
  - –Creare un ‘‘ponte’’ tra sviluppatori, amministratori e utenti
  - –Rilasciare software in modo continuo e diretto (cloud)
  - –Avere frequenti feedback sull’usabilità (User eXperience, UX)

DevOps: processi interni

- **Dev**: sviluppare e testare come in un ambiente di produzione (catena di montaggio)
- **Release**: distribuire con processi automatici
- **Ops**: monitorare la qualità operativa
- Ritorno di **feedback** da parte di utenti, addetti alla qualità, amministratori

DevOps: processo CI

- Continuous Integration and testing (CI):
  - –Gli sviluppatori integrano il codice diverse volte al giorno tramite repository condiviso (SVN, git,…).
  - –Ad ogni check-in
    - Il codice viene assemblato da uno strumento di build automatico per rilevare eventuali problemi.
    - Si eseguono in modo automatico test funzionali e non funzionali (prestazioni, sicurezza, ecc.)
      - –Tutti i test-case oppure solo quelli con priorità più alta

DevOps: processi CD e CO

- **Continuous Delivery e deployment (CD**)

–L’obiettivo è quello di costruire il software in modo da rilasciarlo in qualsiasi momento (production-ready).

–Ogni singola modifica che passa con successo i test viene automaticamente distribuita (cloud).

- **Continuous Operations (CO)**

–Utilizzo da parte degli utenti finali

- Le modifiche software o hardware non devono interrompere l’uso del software da parte degli utenti.

–Monitoring del sistema da parte degli amministratori di sistema e addetti alla qualità

DevOps: processo CA

- **Continuous Assessment (CA)**

–**Feedback loops:** continuo monitoraggio dello stato e delle prestazioni, registrando l'esperienza degli utenti e informando amministratori e sviluppatori.

- ottimizzazione continua del software
- aumento della soddisfazione d'uso (UX) degli utenti finali

–**Planning prioritization:** ai feedback (nuove funzioni o bug-fixing) viene data una priorità in base alle esigenze di business e alle richieste degli utenti finali.

–**Portfolio investment:** quando si ricevono i feedback il gruppo di pianificazione assegna una priorità anche in termini di investimenti necessari.

DevOps: attributi

- **Visibilità media**:
  - –non si conoscono il numero e frequenza delle release
  - –sviluppo organizzato come una catena di montaggio
- **Affidabilità alta:**
  - –Integrazione e collaudo automatico di ogni commit
- **Robustezza alta**:
  - –feedback numerosi e frequenti da parte degli utenti
  - –monitoring di amministratori e addetti alla qualità
- **Rapidità alta:**
  - –software rilasciato in modo continuo e automatico