INGEGNERIA DEL SOFTWARE

Introduzione.  
Cosa si intende per “**ingegneria del software”**?

L’**ingegneria del software** è quella disciplina che si occupa dello **sviluppo**, del **mantenimento** e della **gestione** dei software.

Essa si basa su principi scientifici, metodi e tecniche per progettare, costruire testare e mantenere sistemi software complessi e affidabili.

Gli ingegneri del software adottano un processo sistematica che comprende diverse fai, come l’analisi dei requisiti, la progettazione, l’implementazione, il testing e la manutenzione.

Programmi e software.  
Spesso il termine “**software**” viene utilizzato per indicare un programma per computer. Nell’ambito dell’ingegneria del software tuttavia il termine “software” non identifica solamente i programmi, ma anche tutta la **documentazione** associata, le **librerie**, i **siti** **web** di supporto e i **dati di** **configurazione** che sono richiesti per rendere utili i programmi.

(Se si sviluppa del software che verrà poi utilizzato da altre persone o che sarà modificato da altri ingegneri è importante e utile fornire delle informazioni aggiuntive oltre al codice del programma)

PROCESSI SOFTWARE

Processo software.  
Cosa si intende per “**processo software**”?

Un **processo** **software** è un insieme di attività che porta alla creazione di un software.

Nonostante esistano processi software differenti, ci sono quattro attività comuni a tutti i processi:

* **Specifica del progetto**: cliente e ingegnere colloquiano per definire le funzionalità del software;
* **Sviluppo del software**: viene realizzato il software che realizza i requisiti definiti nella specifica;
* **Convalida del software**: il software deve essere convalidato per garantire ciò che il cliente aveva richiesto;
* **Evoluzione del software**: il software potrà subire dei cambiamenti per soddisfare le necessità dell’utente;

Questa quattro attività al loro interno comprendono ulteriori sotto-attività, nel quale non solo vengono descritti e analizzati gli stati del progetto, ma anche le persone coinvolte nello stesso.

Modelli di processi software.  
I **modelli di processi** **software** rappresentano metodologie organizzative per affrontare lo sviluppo del software.

Ogni modello offre uno specifico approccio alla gestione delle attività di sviluppo.

I cinque modelli più diffusi sono:

* **Il modello a cascata**;
* **Il modello a spirale**;
* **Il Model – Drive Development**;
* **Il RUP**;
* **Il modello Agile.**

Modello a cascata.  
Il **modello a cascata** è il modello più tradizionale.

Il modello a cascata suddivide il processo di sviluppo software in una **sequenza lineare** di fasi **ben definite**, ognuna delle quali deve essere **completata prima che la successiva possa iniziare**.

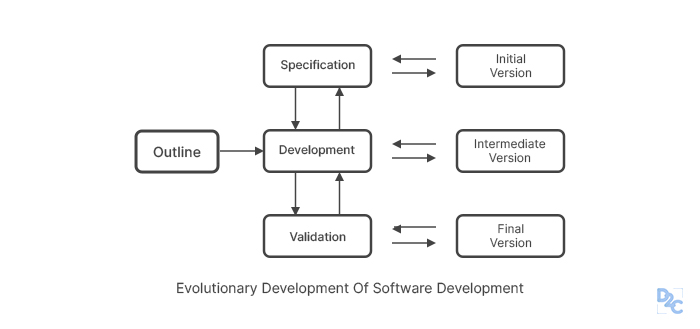
Le fasi principale di questo modello sono:

1. **Analisi e definizione dei requisiti**: si determinano dettagliatamente i servizi del sistema e tutti i suoi vincoli;
2. **Progettazione del sistema e del software**: vengono utilizzati i requisiti raccolti per progettare l’architettura hardware e software del progetto;
3. **Implementazione**: vengono realizzati un insieme di programmi e testati separatamente per controllare che soddisfino le specifiche;
4. **Integrazione e test**: le singole unità prodotte vengono integrate e testate come un unico grande sistema per accertarsi che i requisiti del software siano soddisfatti.
5. **Rilascio e manutenzione**: il sistema viene ufficialmente messo in funzione. Nel corso del tempo vengono corretti gli errori scoperti e aggiunti dei nuovi servici qualora richiesti.

|  |  |
| --- | --- |
| Vantaggi | Svantaggi |
| Struttura del processo chiara | Poco flessibile al cambiamento |
| Documentazione completa | Possibili ritardi sulla consegna |
| Approccio sequenziale | Rischio di insoddisfazione del cliente |
| Controllo rigido su ogni fase | Scarsa adattabilità al cambiamento |

Sviluppo incrementale.  
Lo **sviluppo incrementale** si basa sull’idea di sviluppare un’**implementazione iniziale**, **esporla** agli utenti e perfezionarla attraverso **molte versioni**, finché non si ottiene il prodotto richiesto.

Le varie attività sono intrecciare tra di loro, con feedback veloci tra di esse:



Con questo tipo di approccio si cerca di minimizzare i rischi, consegnare il prodotto richiesto il più velocemente possibile attraverso una più crescente adattabilità ai cambiamenti richiesti dagli utenti.

Il problema di questo approccio potrebbe essere un eccessivo bisogno di test per controllare ogni volta gli aggiornamenti inseriti.

Modello a spirale.  
Il **modello a spirale** è un modello di sviluppo che combina elementi di sviluppo sequenziale (come il modello a cascata) con approcci di tipo incrementale.

Il modello a spirale è caratterizzato da un ciclo di sviluppo a forma di spirale, dove le attività vengono svolte passo dopo passo ma in tempi più brevi, per poi ripartire dall’inizio.

Le fasi del modello a spirale sono:

* **Pianificazione**: il team definisce gli obiettivi del progetto, identifica possibili alternative di sviluppo e considera i vincoli, come budget e scadenza.
* **Analisi e rischio**: il team identifica e valuta i rischi che potrebbero influire sul processo. Si sviluppa quindi un piano per affrontare e mitigare questi rischi;
* **Progettazione e sviluppo**: il team crea il software. Più precisamente in questa fase viene creato un prototipo o una parte del software che verrà nelle iterazioni successive migliorato e incrementato.
* **Valutazione e pianificazione**: il software viene valutato, i risultati vengono esaminati e il team prende decisioni sulla direzione futura. Si decide quindi se iniziare o meno un nuovo ciclo.

|  |  |
| --- | --- |
| Vantaggi | Svantaggi |
| Gestione dei rischi | Complessità |
| Adattabilità | Rischi di costi aggiuntivi |
| Consegne incrementali | Documentazione eccessiva |

RUP

Rational Unified Process (RUP).  
Negli anni sono stati fatti vari tentativi per sviluppare modelli di processi “universali”. Uno dei più noti di questi modelli universali è il “**RUP**” (**R**ational **U**nified **P**rocess).

Il **RUP** è un **framework di sviluppo software**, ovvero un **insieme** **organizzato** e **strutturato** di **linee guida**, **processi**, **procedure**, **modelli** e **strumenti** che forniscono una **struttura** per la **gestione** del ciclo di vita del software.

Il altre parole il RUP **non è un processo di sviluppo software specifico** o un metodo dettagliato, ma piuttosto un **quadro concettuale** **flessibile** che può essere **adattato** e **personalizzato** per soddisfare le esigenze dei progetti.

Le fasi del RUP.  
Nel RUP il ciclo di sviluppo del software comprende le seguenti fasi:

* **Fase iniziale** (**inception** **phase**);
* **Fase di elaborazione** (**elaboration phase**);
* **Fase di costruzione** (**costruction phase**);
* **Fase di transizione** (**transition phase**);

Ogni fase ha un insieme di obiettivi e si conclude con la realizzazione di una “**milestone**”(prodotto) di qualche genere. Le “**milestone**” sono dei punti di controllo che permettono di valutare lo stato del processo e la direzione da prendere.

Fase iniziale.  
In questa fase vengono **definiti gli obiettivi**, si **identificano i rischi** iniziali, si stabiliscela **fattibilità** del sistema e si **stimano** i **tempi** e i **costi** per produrre il software.

Fase di elaborazione.  
Durante questa fase, si svolge una **progettazione dettagliata** del sistema e si **stabilisce l’architettura**. Si **stabiliscono** e si **affrontano** i **rischi** principali. I requisiti vengono analizzati e finalizzati.

Fase di costruzione.  
Questa è la **fase principale di sviluppo**, in cui il codice viene effettivamente costruito. Vengono creati componenti software, aggiunte funzionalità in modo incrementale e vengono effettuati test continui. L’obiettivo di questa fase è quella di **costruire una prima implementazione** del sistema funzionante.

Fase di transizione.  
Questa fase prevede la **realizzazione** e la **consegna** del **prodotto finale**. Si effettuano i test di accettazione per vedere se il prodotto rispetta quanto concordato nella fase iniziale e vengono condotte le attività di training dei clienti.

Aspetti statici del RUP.  
Il RUP, per definire e controllare le varie fase del processo di sviluppo, utilizza i cosiddetti “**concetti statici**”, ovvero che sono definiti nello stesso modo per tutti i processi. Questi concetti sono:

* **Ruoli**(**chi**): i ruoli identificano il **comportamento** e le **responsabilità** di un **individuo** o di un **gruppo** di individui;
* **Attività** (**come**): sono **compiti specifici**, come l’analizzare, l’elaborare o il revisionare, che i membri del team **svolgono** per portare avanti il processi di sviluppo;
* **Elaborati** (**cosa**): rappresentano i **risultati tangibili delle attività** svolte all’interno del processo, come ad esempio modelli, documenti, codice sorgente o eseguibile;
* **Workflow** (**quando**): rappresenta la **sequenza temporale delle attività** e degli **eventi** all’internodel processo di sviluppo. Viene quindi descritto come le attività vengono eseguite, chi è il responsabile di ciascuna attività e come interagiscono tra di loro all’interno del processo.

(Notiamo bene quindi come ogni aspetto statico risponda ad una domanda fondamentale.   
I ruoli corrispondono al “chi” fa qualcosa.  
Le attività corrispondono al “come” un individuo fa qualcosa.  
Gli elaborati corrispondono al “cosa” un individuo riesce a produrre.  
Il workflow corrisponde al “quando” un individuo produce un qualcosa.)

|  |  |
| --- | --- |
| Vantaggi | Svantaggi |
| Feedback regolare con gli utenti | Potenziale complessità |
| Utilizzo efficiente delle risorse | Necessità di persone esperte |
| Consegna nei tempi |  |
| Scoperta precoce dei rischi |  |

SVILUPPO AGILE DEL SOFTWARE

Sviluppo Agile.  
Lo “**sviluppo agile**” indica un **insieme di metodi** di sviluppo software emerso come risposta alle sfide incontrate nello sviluppo del software all’interno di un ambiente in rapida evoluzione.

È nato come alternativa di modelli che spesso risultavano rigidi, poco adattabili ai cambiamenti e generavano ritardi nelle consegne.

I valori sui cui si basa una metodologia agile che segua i punti indicati nel **Manifesto Agile** sono quattro:

* **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**.

|  |  |
| --- | --- |
| Vantaggi | Svantaggi |
| Flessibilità | Complessità gestionale |
| Consegna continua | Documentazione limitata |
| Coinvolgimento del cliente |  |

Nel tempo sono stati creati vari modelli che si basano su un approccio agile, ma i più noti sono:

* **La programmazione estrema**;
* **Scrum**.

Programmazione estrema.  
La **programmazione estrema** (**XP**, e**X**streme **P**rogramming) è un modello che tende a spingere le normali pratiche a livelli estremi.

Nella programmazione estrema, i requisiti sono espressi come scenari, le cosiddette “**storie utente**”. Una storia utente non è altro che uno scenario in cui potrebbe trovarsi un utente del sistema durante l’utilizzo dello stesso.

In questo modello i **programmatori lavorano a coppie** e **sviluppano test per ogni compito ancora prima di scrivere codice**. Prima di integrare del nuovo codice nel sistema, tutti i test devono essere superati.

Il ciclo di sviluppo della programmazione estrema prevede le seguenti fasi:

* **Selezione delle storie utente**: vengono identificate le storie utente, e quindi le funzionalità, che saranno incluse nella prossima release;
* **Suddivisione delle storie in task**: le storie utente vengono suddivise in task più piccole e gestibili;
* **Pianificazione della release**: il team pianifica la release, cioè la versione del software che verrà rilasciata al cliente;
* **Sviluppo**, **integrazione** e **test** **del software**: viene sviluppato il codice e ogni membro del team lavora sulle task assegnate. Il codice sviluppato viene integrato e vengono eseguiti i test per verificare che il software funzioni correttamente;
* **Release del software**: il software è pronto e funzionante, quindi viene rilasciato;
* **Valutazione del sistema**: il software viene valutato, vengono raccolti i feedback dagli utenti ed eventuali problemi vengono identificati (per poi essere corretti nella prossima release).

Queste fasi vengono ripetute ciclicamente fino al completamento del progetto.

Refactoring.  
Per facilitare le modifiche da apportare al codice, gli sviluppatori XP suggeriscono che il codice che si sta sviluppando debba essere costantemente **rifattorizzato**. La **rifattorizzazione** (**refactoring**) richiede che il team si programmazione ricerchi possibili miglioramenti del software e li implementi immediatamente.

Esempi di refactoring sono:

* Riorganizzazione della gerarchia delle classi per eliminare il codice duplicato;
* Riordino e ridenominazione degli attributi e dei metodi;

Sviluppo con test iniziali.  
La programmazione estrema ha sviluppato un nuovo approccio al testing dei programmi: anziché scrivere il codice e poi il test per il codice, si **scrive prima il test e poi il codice**. Questo significa che si possono eseguire i test mentre di sta scrivendo il codice e quindi scoprire eventuali problemi durante lo sviluppo.

Scrum.  
**Scrum** è un **framework** per **organizzare progetti agile** e fornisce, in una certa misura, una visibilità esterna su ciò che sta accadendo.

Il ciclo di sviluppo è suddiviso in “**sprint**”, ovvero un **arco di tempo** in cui **viene prodotto** **parte del software**. Alla fine di ogni sprint è prevista la consegna incrementale del software in modo da riceve feedback dal cliente o dagli stakeholder.

Il processo Scrum si articola di 3 componenti principali: **Ruoli**, **Artefatti** e **Eventi**.

Ruoli.  
Scrum identifica tre ruoli specifici:

* **Product** **owner**: ha il compito di **identificare** le **caratteristiche** o i **requisiti** del **prodotto**, stabilirne le **priorità** e rivedere continuamente il **product backlog** per garantire che il progetto continui a soddisfare i requisiti;
* **Scrum master**: è il **responsabile del processo**. Ha il ruolo di **coordinare** e **mettere** **in condizione il team di lavorare** in maniera **coerente** con lo sviluppo del progetto, **eliminando** eventuali **ostacoli** che impattino sulla produttività;
* **Team di sviluppo**: gruppo di sviluppatori **responsabili** **dello sviluppo** **del** **software** e di altri documenti essenziali per la progettazione.

Artefatti.  
Gli Scrum team utilizzano strumenti chiamati “**artefatti**” per la risoluzione e la gestione di progetti. Gli **artefatti** offrono sia ai membri del team che alle parti interessate, informazioni essenziali riguardo la pianificazione e le attività.

Ci sono 3 tipi di artefatti:

* **Product backlog**: è il documento che contiene la **lista di tutti i requisiti necessari** per la realizzazione del progetto.
* **Sprint backlog**: è il documento che definisce tut**ti i task da completare** nei singoli sprint.
* **Increments**: è un documento che contiene **tutti i task completati durante un sprint**.

Eventi.  
Gli **eventi** non sono altro che delle **riunioni svolte regolarmente**, volte a **controllare** **l’andamento** del progetto, creare **uniformità** e **sincronizzare** le attività.

Ci sono 4 tipi di eventi:

* **Sprint Planning**: riunione in cui il **product owner stila il product backlog**, in presenza del team e dello scrum master. **Al termine della** **riunione**, il product owner può **compilare lo sprint backlog**.
* **Daily Scrum**: riunione tra lo scrum master e il team. Durante questa riunione **viene annotato il lavoro svolto il giorno precedente e creato un piano per le prossime 24 ore**. Questo serve a sincronizzare le attività.
* **Sprint Review**: riunione che viene fatta al termine di ogni sprint per **verificare le attività svolte per il raggiungimento degli obiettivi fissati in quello sprint**.
* **Sprint Retrospective**: riunione in cui viene effettuata **un’analisi del progetto**, delle **attività** e degli **obiettivi**. L’obiettivo di questa riunione è di **perfezionare costantemente** sia il **processo** che la **realizzazione** **del prodotto**. Questa riunione prevede anche la partecipazione degli utenti finali.

PROJECT PLANNING

Project Planning.  
Il **project planning**, o **pianificazione del progetto**, è il processo di **definizione**, **organizzazione** e **gestione** delle attività necessarie per portare a termine con successo un progetto software.

Le principali fasi che compongono un project planning sono:

* **L’accordo**;
* **La pianificazione**;
* **L’azione**;
* **La rivisitazione**.

Ognuna di queste fasi contiene al suo interno delle sotto-attività.

L’accordo.  
**L’accordo** è il punto di partenza per qualsiasi progetto software. Questa fase riguarda la **definizione degli obiettivi**, aspettative, **requisiti** e **vincoli** che saranno poi formalizzati in un contratto una volta che le parti sono d’accordo. Il contratto si chiama “Process initiation Document”.

Tempo, qualità e costi.  
Durante la stipulazione di un contratto ci sono tre grandezze fondamentali da tenere in considerazione:

* **Il tempo;**
* **La** **qualità**;
* **Il** **costo**;

Queste grandezze sono tra di loro correlate e al variare di una corrisponde una variazione dell’altra. Possono essere direttamente o inversamente proporzionali in base al caso che si sta trattando.

(ad esempio, è facile intuire che se viene richiesta un’alta **qualità** del software in poco **tempo**, allora il **costo** della progettazione si alzerà)

Pianificazione.  
La fase di **pianificazione** serve a **definire** **come il progetto sarà gestito dall’inizio alla fine**.

Questa fase comprende svariate sotto-attività come:

* **Lista dei task**;
* **Assemblamento del team**;
* **Stime del tempo e dei costi**;
* **Suddivisione del lavoro** (costruzione del **PERT** e del **critical path**);
* **Rilevamento e gestione dei rischi**.

Pianificazione: lista dei tesk.  
Questa fase consiste nell’analizzare e stabile gli obiettivi del progetto. Le attività vengono divisi in sotto-attività, le task, più piccole e più gestibili.

Pianificazione: assemblamento del team.  
In questa fase ci costruisce il team che parteciperà alla realizzazione del prodotto. La scelta dei componenti può variare in base al tipo di progetto, ma alcune figure sono molto comuni, come ad esempio l’amministratore del database, sviluppatore front-end, sviluppatore back-end, ecc….

Stima dei tempi e dei costi.  
In questa fase viene fatta una stima riguardo i tempi e i costi di realizzazione del progetto.

L’analisi del tempo e dei costi può essere ottenuta attraverso metodi informali che formali:

|  |  |
| --- | --- |
| Informale | Formale |
| Giudizio di un esperto: il costo viene stimato tramite l’esperienza di un esperto. | Algoritmi: il costo ci è dato da una formula:  COSTO = A x SIZEB x M  Dove:  A = valore costante che dipende dall’organizzazione interna e dal tipo di software;  SIZE= dimensioni del software in senso di linee di codice.  B = riflette la non linearità dei costi in relazione alle dimensioni del software. Se la taglia del progetto aumenta, vengono aggiunti sempre più costi extra. Solitamente è un valore compreso tra 1 e 1.5.  M = moltiplicatore che tiene conto degli attributi legati al processo, al prodotto e all’esperienza del team di sviluppo. I fattori in questo sono ad esempio l’affidabilità, la complessità del progetto e l’esperienza del team. |
| Analogia: il costo viene stimato mettendolo in relazione con un caso simile di cui si già conosce una stima di costo. |
| Metodo Delphi: più esperti esprimo in modo indipendente una stima.  Un mediatore, distribuirà in maniera anonima le stime tra tutti gli esperti per poi ripetere nuovamente la procedura.  Si continua fino a quando non si è tutti d’accordo. |

Pianificazione: costruzione del PERT e del critical path.  
Una volta stilata una lista di task, viene effettuata una stima per la realizzazione di ognuna di esse.

Una volta attribuita una durata ad ogni task, si passa alla costruzione del PERT e all’individuazione del critical path.

Il **PERT** (**Program Evaluation and Review Technique**) è una **tecnica di gestione dei progetti** utilizzata per **pianificare le attività complesse**.

In termini più pratici il PERT non è altro che un grafo dove i vertici sono i task individuati e le frecce che collegano i vertici sono le eventuali dipendenze tra le attività.

Grazie al PERT è possibile ottenere due informazioni importanti circa il progetto che si sta sviluppando:

* Il **complition time** (**tempo di completamento**): **tempo necessario a sviluppare il progetto**;
* Il **critical path** (**cammino critico**): un cammino in cui sono presenti tutte le attività il cui ritardo nel completamento potrebbe causare un **ritardo generale** nella consegna del progetto.

Facciamo un esempio per capire meglio. Supponiamo di avere le seguente attività con le relative stime di tempo e dipendenze:

|  |  |  |
| --- | --- | --- |
| Attività (task) | Predecessore | Durata |
| A | - | 3 |
| B | A | 4 |
| C | A | 2 |
| D | B | 5 |
| E | C | 1 |
| F | C | 2 |
| G | D,E | 4 |
| H | F,G | 3 |

(Questa tabella viene chiamata “matrice delle dipendenze”)

**AVVERTENZA**: se due attività non sono collegate, come ad esempio l’attività “B” e l’attività “C”, possono essere eseguite in parallelo? La risposta è SI! Se si hanno le risorse disponibili allora è possibile eseguire due attività contemporaneamente.

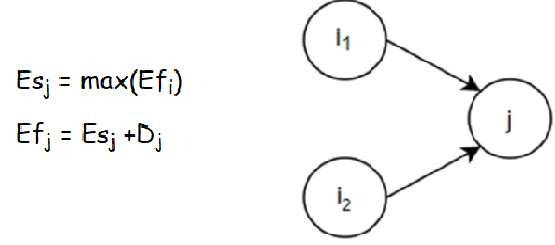
Arrivati a questo punto, andiamo a calcolare la prima informazione che ci interessa: il **completion time**!

Per calcolare il **completion time**, ovvero il **tempo necessario per completare il progetto**, bisogna intuitivamente andare a calcolare **quanto tempo ci vuole per terminare ogni singola attività**. Bisogna quindi andare a calcolare quello che viene chiamato “**Forward Path**”.

Per calcolare il **Forward Path** bisogna andare a stabilire l’**Earl Start** (**E**s) e l’**Earl Finish** (**Ef**) di ogni task. Questi due termini rappresentano rispettivamente:

* **Earl Start**: indica quanto prima si può iniziare un’attività;
* **Earl Finish**: indica quanto prima si può terminare un’attività.

Come faccio a calcolare quindi questi due parametri? Bisogna seguire queste due regolette:



Vediamolo in pratica sul nostro esempio:

Ogni nodo presente nel grafo creato precedentemente assumerà adesso la seguente forma:

|  |  |
| --- | --- |
| nome del nodo | |
| Earl Start | Earl Finish |
| Late Start | Late Finish |

(Vedremo successivamente a cosa servono Late Start e Late Finish)

Partiamo dall’attività “A”. L’attività “A” è la prima attività che viene iniziata. Non avendo quindi predecessori, possiamo dire che inizierà al tempo 0.

Quindi EsA = 0.

|  |  |
| --- | --- |
| A | |
| 0 | Earl Finish |
| - | - |

Quant’è il tempo di durata (D) dell’attività “A”?   
Andiamo a controllare nella matrice delle dipendenze e vediamo che è 3.

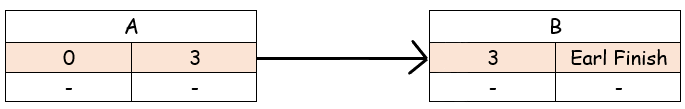
Quindi EfA sarà uguale a EsA + D, ovvero: 0 + 3 = **3**.

|  |  |
| --- | --- |
| A | |
| 0 | 3 |
| - | - |

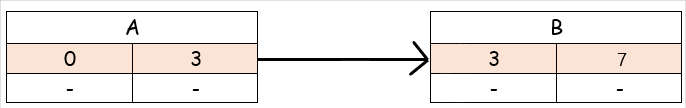
Adesso che abbiamo fatto l’attività “A” possiamo passare a fare l’attività “B” o “C”. Procediamo arbitrariamente con “B”.

NOTA: “B” è la seconda attività, quindi per calcolare il suo Es dobbiamo seguire la regola fornita nella pagina precedente.

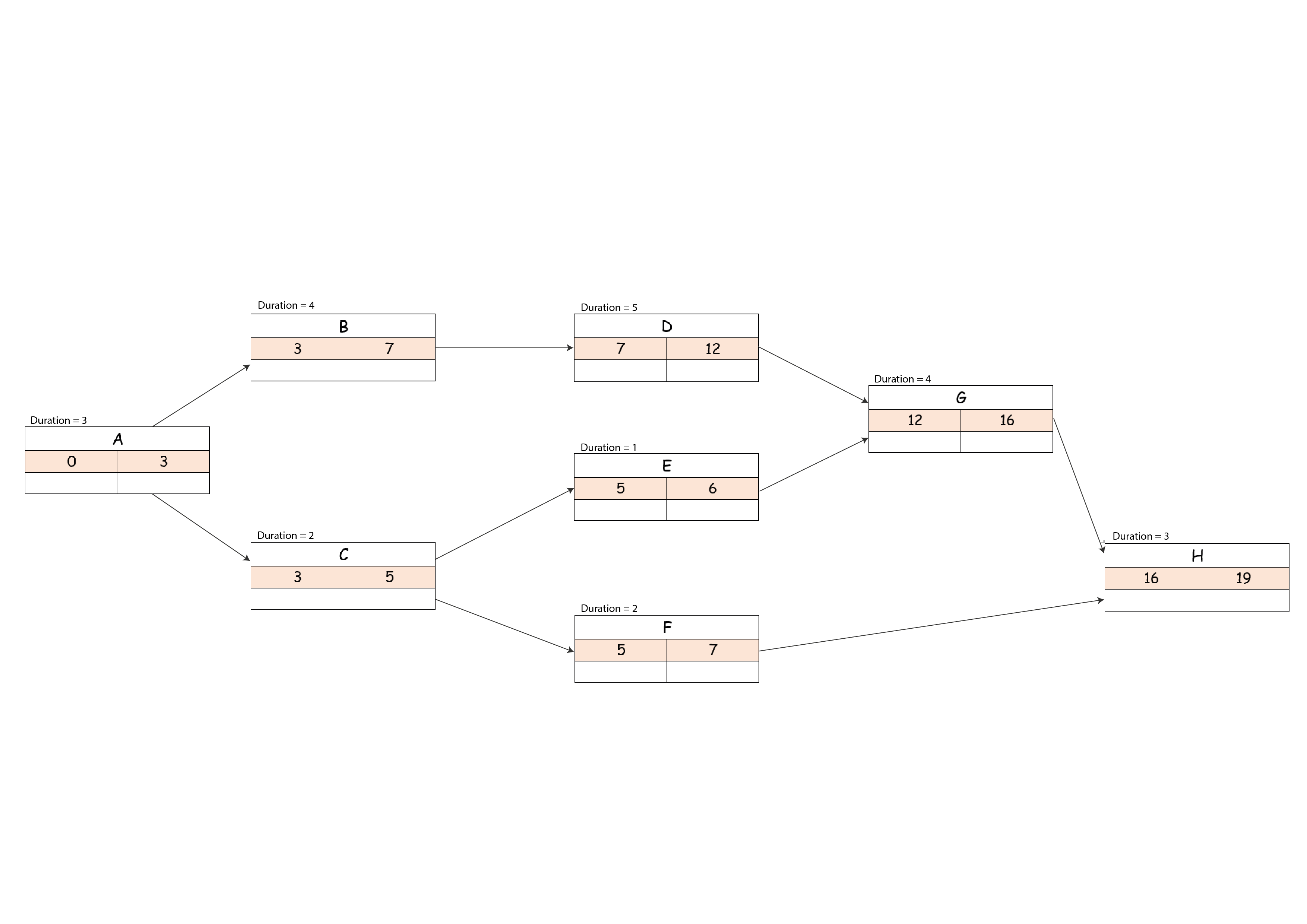
EsB sarà quindi uguale al più grande Es dei suoi predecessori. In questo caso “B” ha solo un predecessore (ovvero “A”), quindi EsB = EsA .



EfB, come fatto prima per “A”, sarà uguale a EsB + DB, ovvero: 3 + 4 = **7**.

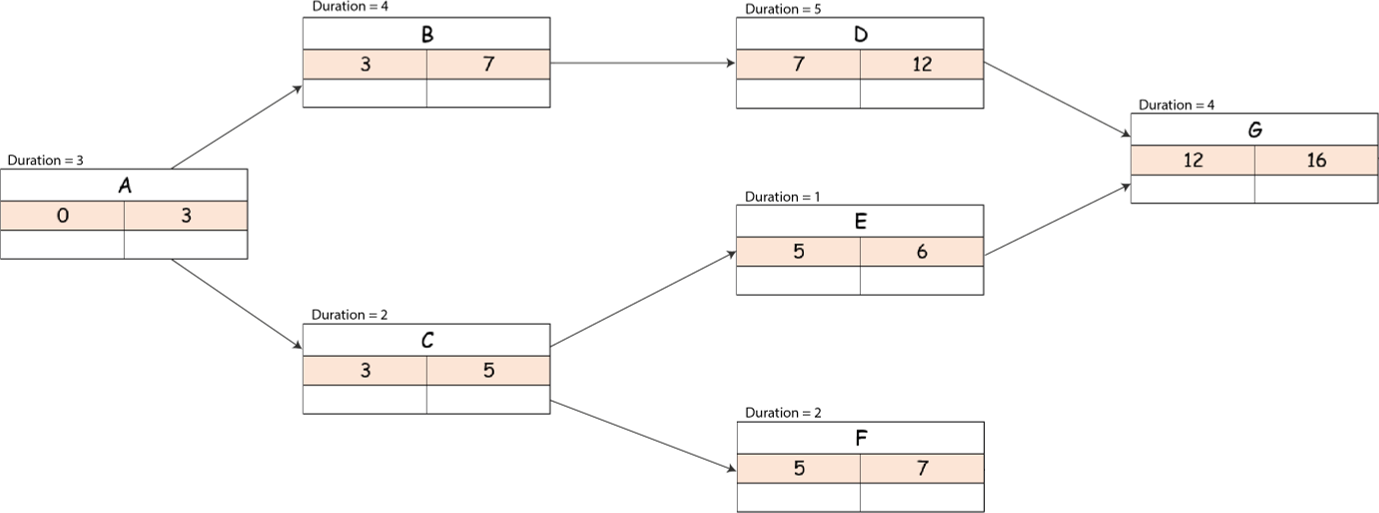


Continuiamo adesso così per l’attività “C”, “D”, “E” e “F” per poi vedere una casistica particolare in cui ci possiamo imbattere in questo procedimento.



Analizziamo adesso l’attività “G”. L’attività “G” ha due predecessori, quindi per la regola che abbiamo fornito, dobbiamo prendere Es più grande tra i suoi predecessori.

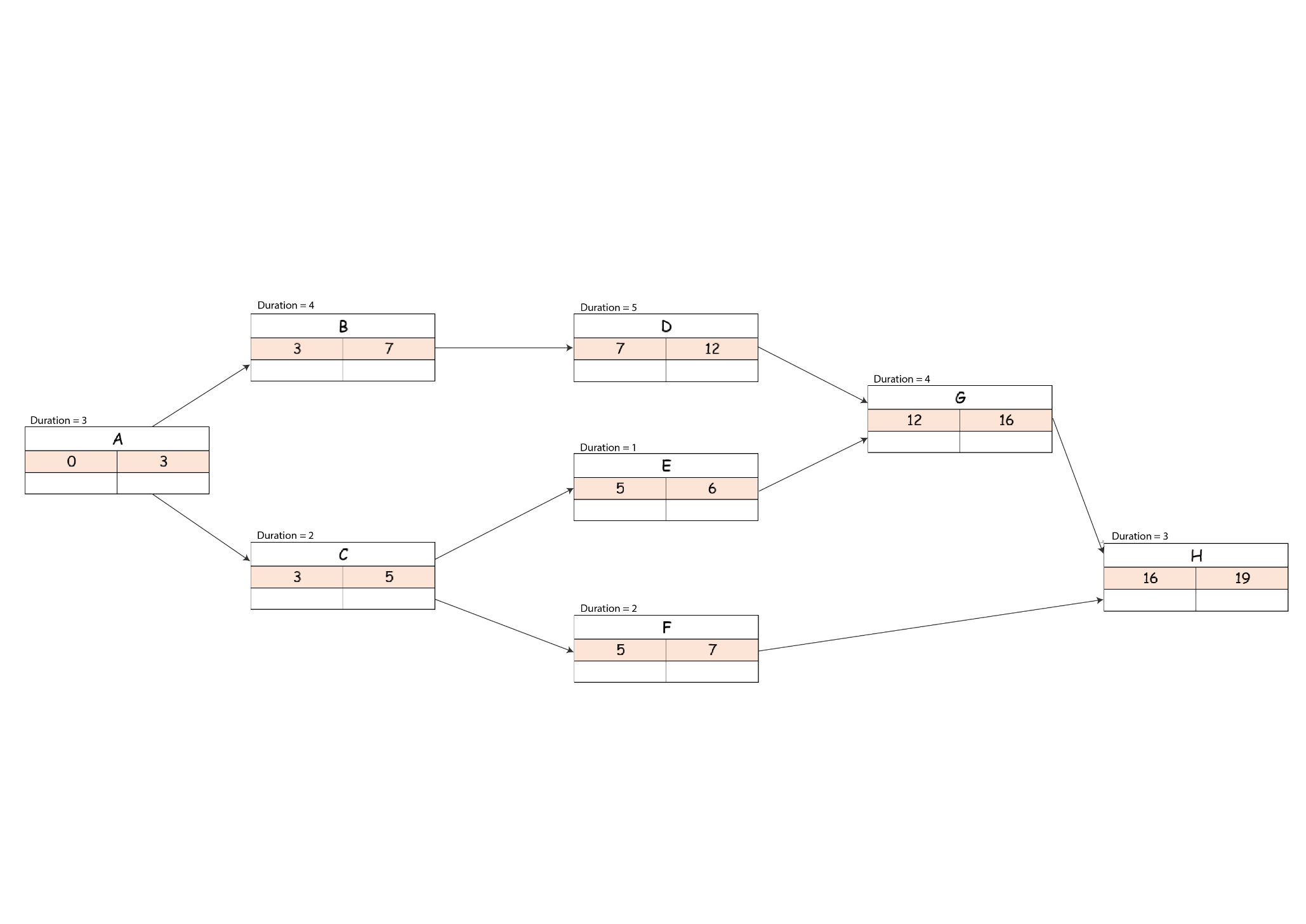
Tra l’attività “E” e l’attività “D”, EsD è maggiore di EsE, quindi EsG = EsE.



Stesso identico ragionamento per l’attività “H”.

Dal momento che EsG è maggiore di EsF, allora EsH = EsG.

(Immagine presente nella pagina successiva)



Una volta terminato il Forward Path, ci basterà andare a vedere EfH, ovvero **l’ultima attività effettuata** per ottenere il completion time.

Nel nostro esempio, **il completion time** = 19.

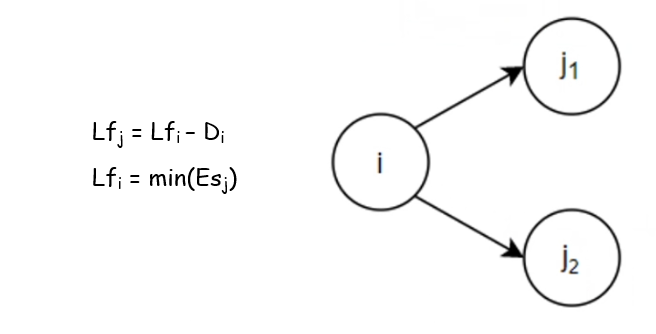
Possiamo adesso andare a calcolare la seconda informazione che ci interessa: il **critical path**!

Per calcolare il **critical path**, ovvero tutte quelle attività (definite per l’appunto “critiche”) il cui ritardo, anche di una sola, potrebbe generare un **ritardo nel completamento del progetto**, bisogna ripercorrere il grafo all’indietro e andare a calcolare quello che viene chiamato “**Backward Path**”.

Per calcolare il **Backward Path** bisogna andare a stabilire il **Late Start** (**L**s) e il **Late Finish** (**Lf**) di ogni task. Questi due termini rappresentano rispettivamente:

* **Late Start**: indica quanto più tardi si può iniziare un’attività;
* **Late Finish**: indica quanto più tardi si può terminare un’attività.

Come per il calcolo del Es e Ef, anche qui bisogna seguire due regole:



(già dalla loro dichiarazione possiamo vedere come siano il contrario di quelle fornite in precedenze)

Continuiamo adesso con il nostro esempio.

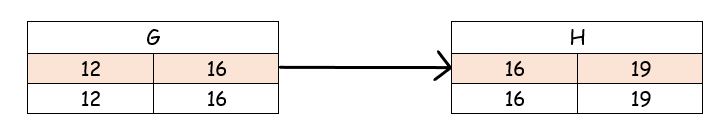
Partiamo dall’attività “H”.  
Essendo la prima attività da cui partiamo per calcolare il critical path, allora il suo LfH sarà uguale al suo EfH.

|  |  |
| --- | --- |
| H | |
| 16 | 19 |
| - | 19 |

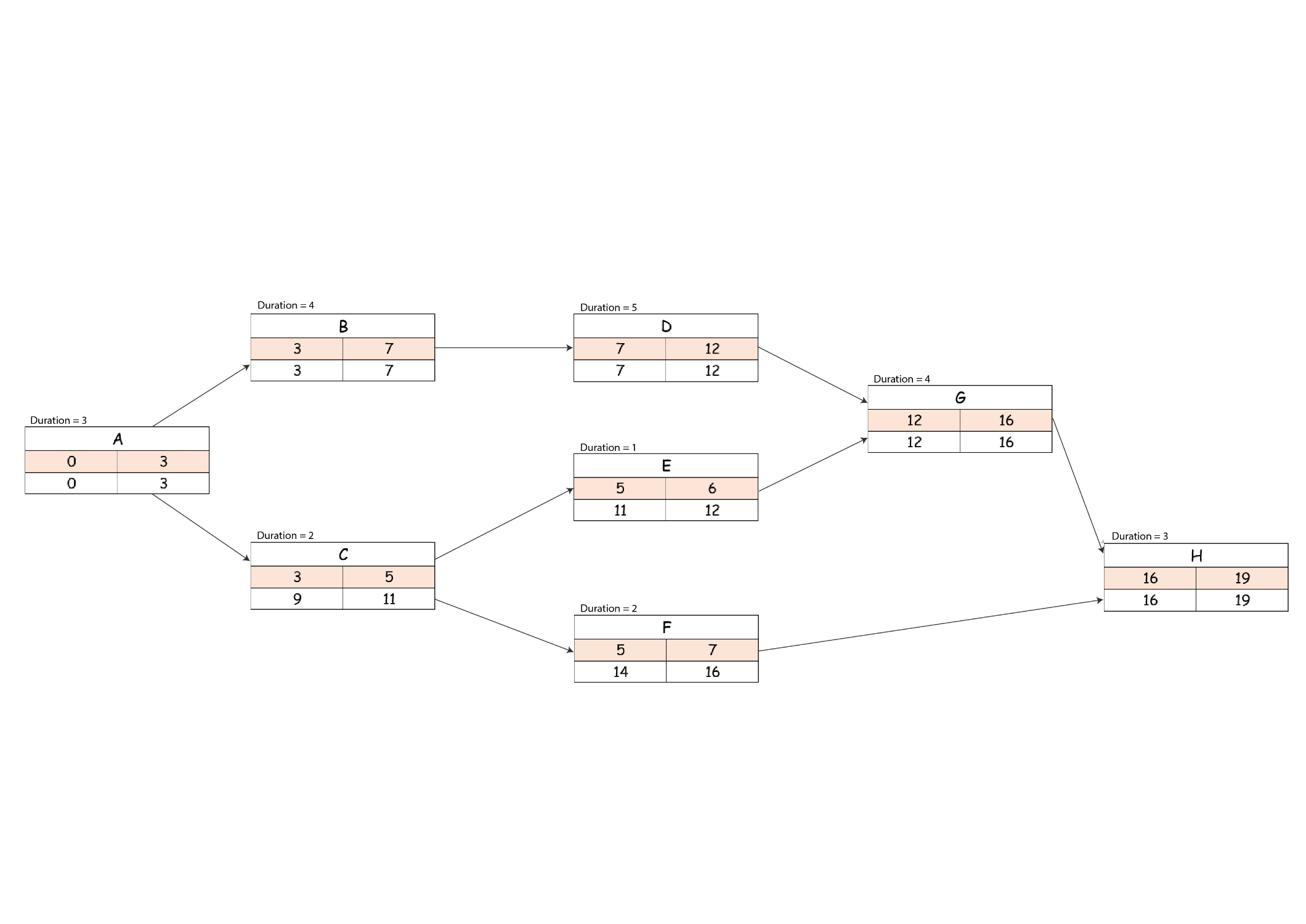
Il suo LsH sarà uguale al suo LfH, quindi 19, meno la sua durata, quindi 3. Quindi abbiamo LFH – DH, ovvero 19 – 3 = 16.

|  |  |
| --- | --- |
| H | |
| 16 | 19 |
| 16 | 19 |

Proseguiamo con l’attività “G”.   
LfG sarà uguale a LsH, quindi -> LfG = LsH -> 16.  
LsG sarà uguale a LfG – DG -> 16 – 4 = 12.



Continuiamo così con tutte le attività ricordando che quando un’attività ha due successori (come ad esempio l’attività “C” che ha come successori l’attività “E” e “F” e l’attività “A” che ha come successori “B” e “C”) bisogna considerare Ls più piccolo.



Una volta completato il Backward Path, andiamo a calcolare per ogni attività il **Total Float**!

Cos’è il **Total Float**?

Per **Total Float** si intende:

**Ls – Es  
Lf – Ef**

Se entrambe queste differenze ritornano il risultato 0, allora vuol dire che quell’attività è un’attività critica.

Andiamo ad esempio a calcolare il Total Float dell’attività “A”:

|  |  |
| --- | --- |
| A | |
| 0 | 3 |
| 0 | 3 |

Abbiamo:

Ls – Es -> 0 – 0 = 0.

Lf – Ef -> 3 – 3 = 0.

Questo vuol dire che “A” è un’attività critica.

Continuiamo così per tutte le attività:

|  |  |
| --- | --- |
| A | |
| 0 | 3 |
| 0 | 3 |

Ls – Es -> 0 – 0 = 0.

Lf – Ef -> 3 – 3 = 0.

|  |  |
| --- | --- |
| B | |
| 3 | 7 |
| 3 | 7 |

Ls – Es -> 3 – 3 = 0.

Lf – Ef -> 7 – 7 = 0.

|  |  |
| --- | --- |
| C | |
| 3 | 5 |
| 9 | 11 |

Ls – Es -> 9 – 3 = 6.

Lf – Ef -> 11 – 5 = 6.

|  |  |
| --- | --- |
| D | |
| 7 | 12 |
| 7 | 12 |

Ls – Es -> 7 – 7 = 0.

Lf – Ef -> 12 – 12 = 0.

|  |  |
| --- | --- |
| E | |
| 5 | 6 |
| 11 | 12 |

Ls – Es -> 11 – 5 = 6.

Lf – Ef -> 12 – 6 = 6.

|  |  |
| --- | --- |
| F | |
| 5 | 7 |
| 14 | 16 |

Ls – Es -> 14 – 5 = 9.

Lf – Ef -> 16 – 7 = 9.

|  |  |
| --- | --- |
| G | |
| 12 | 16 |
| 12 | 16 |

Ls – Es -> 12 – 12= 0.

Lf – Ef -> 16 – 16 = 0.

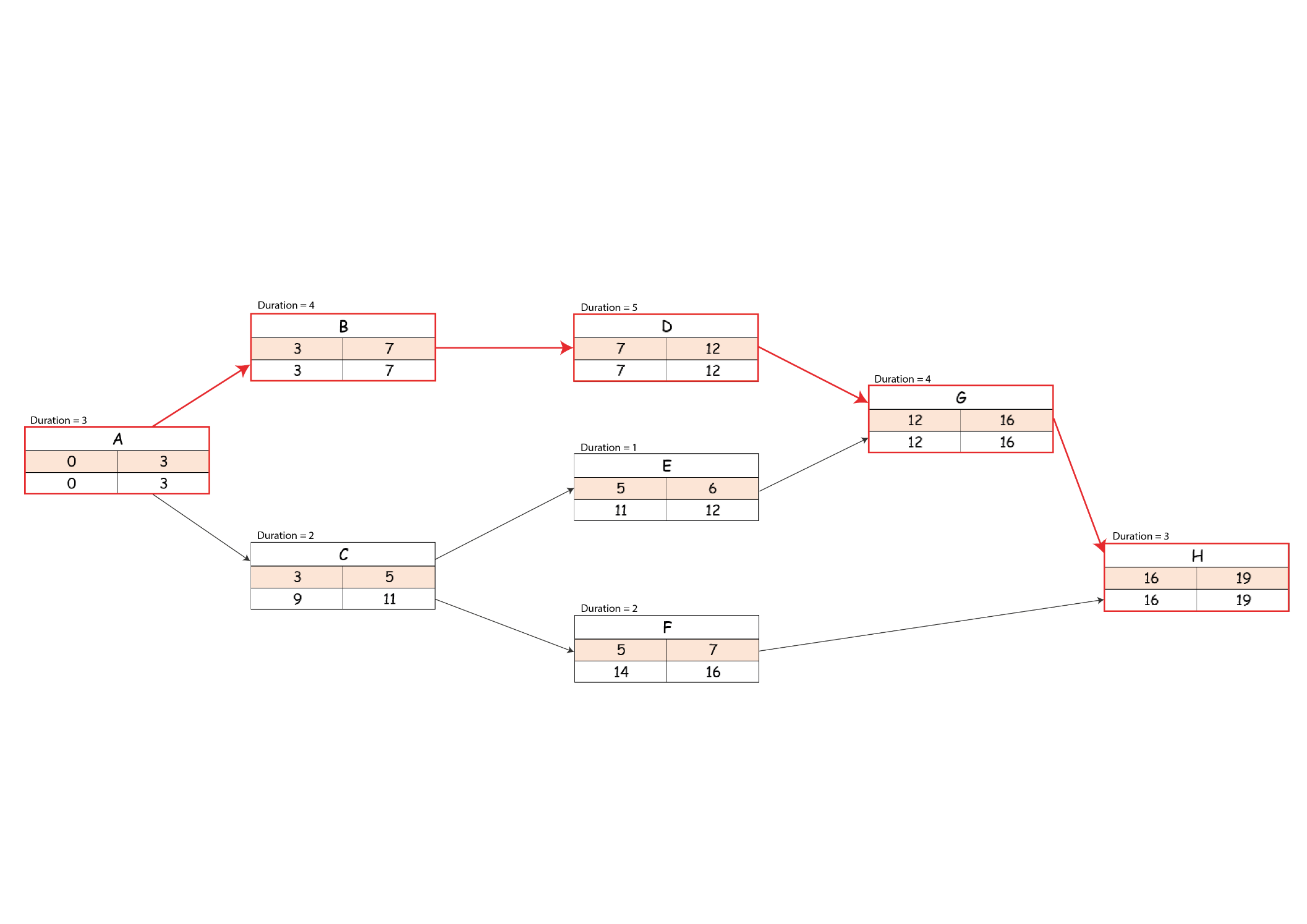
|  |  |
| --- | --- |
| H | |
| 16 | 19 |
| 16 | 19 |

Ls – Es -> 16 – 16= 0.

Lf – Ef -> 19 – 19 = 0.

Nel nostro esempio possiamo vedere che “A”, “B”, “D”, “G” e “H” sono attività critiche, e quindi rappresentano il **critical path**.

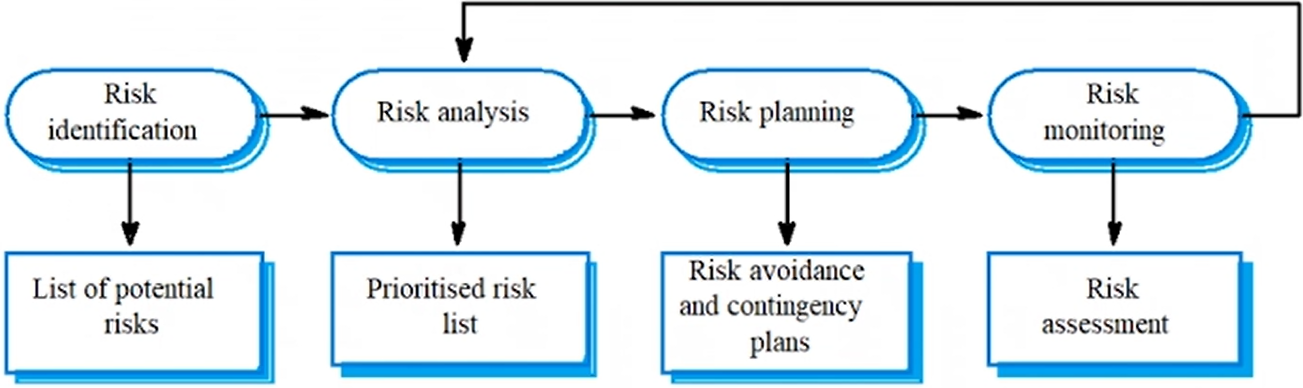
Graficamente rappresentiamo le attività critiche evidenziando le loro frecce di rosso.



Pianificazione: gestione dei rischi.  
In questa fase di cerca di **prevenire**, **monitorare** e **gestire** i **rischi**.

In particolare questa fase si articola in quattro fasi:

1. **Identificazione dei rischi**;
2. **Analisi dei rischi**;
3. **Gestione dei rischi**;
4. **Monitoraggio dei rischi**.



Azione.  
Questa parte comprendere quello che viene detto “**Monitor and Reschedule**”, ovvero **monitorare il progetto** ed **eventualmente riadattarlo qualora ci fosse bisogno**.

Review.  
Una volta terminato qualsiasi progetto è bene analizzare ciò che è andato **bene**, ciò che è andato **male** e ciò che poteva **andare meglio**, in modo da avere un feedback e migliorarsi per il prossimo progetto.

INGEGNERIA DEI REQUISITI

Ingegneria dei requisiti.  
Cos’è un **requisito**?

Un **requisito** è una **descrizione di cosa deve fare il sistema** o un **vincolo** che lo riguarda.

L’ingegneria dei requisiti è quel processo che ci consente di identificare i **servizi** che l’utente richiede, la **qualità** dei servizi e i **vincoli** sotto i quali si deve operare.

All’interno del mondo dell’ingegneria dei requisiti abbiamo due grandi macro-categorie di requisiti:

|  |  |
| --- | --- |
| Requisiti funzionali | Requisiti non funzionali |
| Descrivono cosa il sistema deve fare. | Descrivono le qualità, le proprietà e le caratteristiche che i requisiti funzionali devono avere. |

Requisiti funzionali   
I **requisiti funzionali** descrivono “**COSA**” il sistema deve fare. Si concentrano sulle azioni, le operazioni e le funzioni che il sistema deve svolgere.

Spesso quando si cercano di definire questo tipo di requisiti, ci si imbatte in una serie di difficolta:

* **Ambiguità dei requisiti**: i requisiti potrebbero essere interpretati in differenti modi;
* **Incompletezza**: la lista potrebbe non essere completa;
* **Inconsistenza**: due requisiti potrebbero entrare in contraddizione tra di loro.

Processo di deduzione dei requisiti funzionali.  
Per scoprire quali sono i requisiti funzionali, esiste un processo in cui lo staff tecnico, i clienti e ogni altra figura inerente al progetto si riuniscono e cercano di capire cosa il sistema deve fare.

Il processo di scoperta dei requisiti prevede quattro fasi:

1. **Scoperta dei requisiti;**
2. **Classificazione e organizzazione dei requisiti;**
3. **Prioritizzazione dei requisiti;**
4. **Specifica dei requisiti.**

Scoperta dei requisiti.  
Durante questa fase si cerca si scoprire quali sono i requisiti. Per scoprire quali sono i requisiti ci sono principalmente quattro tecniche informali:

* **Questionari e interviste;**
* **Etnografia;**
* **Storie utente e scenari;**
* **Casi d’uso.**

Scoperta dei requisiti: questionari e interviste.  
I **questionari** e le **interviste** sono due strumenti diversi e che devono essere usati in ambiti diversi:

* Un questionario è un documento in cui sono raccolte delle domande a cui gli utenti dovranno rispondere.
* Un’intervista invece prevede l’interazione diretta con le persone interessate.

|  |  |  |
| --- | --- | --- |
|  | Vantaggi | Svantaggi |
| Questionario | Strumento asincrono che non prevede che le persone siano tutte fisicamente presenti nello stesso luogo e nello stesso momento. | Sforzo maggiore nella scelta delle domande, le quali dovranno essere ben dettagliate. |
| Intervista | Necessità di incontrare fisicamente le persone interessate. | Si possono sviluppare domande e mitigare dubbi nel corso dell’intervista. |

Scoperta dei requisiti: etnografia.  
**L’etnografia** prevede l’analisi delle persone all’interno del loro contesto lavorativo. Uno o più membri dello staff tecnico hanno il compito di osservare fisicamente come si comportano gli utenti e di trarre da questa osservazione i requisiti.

Scoperta dei requisiti: storie utente e scenari.  
Una **storia utente** è una **dichiarazione**, **scritta in linguaggio naturale**, che **descrive** una **funzionalità** dal punto di vista dell’utente. **Racconta una situazione di vita quotidiana in cui l’utente utilizza il sistema**.

Le storie utente sono particolarmente utili in quanto forniscono una sequenzialità degli eventi.

Gli scenari invece sono utilizzati per descrive un po’ più nel dettaglio come un sistema dovrebbe comportarsi in risposta ad un insieme specifico di condizioni o eventi.

In altre parole uno scenario è una descrizione sequenziale delle azioni, degli eventi e delle risposte del sistema in un contesto specifico.

Scoperta dei requisiti: casi d’uso.  
I **casi d’uso** sono un modo di descrivere le interazioni tra utenti e sistema utilizzando un modello grafico e un testo strutturato.

Nella loro forma più semplice, un **caso d’uso identifica gli attori coinvolti in un’interazione e assegna un nome al tipo di interazione**. Poi si aggiungono altre informazioni che descrivono le interazioni con il sistema.

Classificazione e organizzazione dei requisiti.  
La **classifica** e **l’organizzazione** dei requisiti sono quelle attività che, partendo dalla raccolta non strutturata dei requisiti, **raggruppa** quelli **correlati** e li **organizza** in **gruppi** **coerenti**.

Prioritizzazione e negoziazione dei requisiti**.**I requisiti spesso possono essere in conflitto. Quest’attività di occupa di dare una priorità ai requisiti, trovare e risolverne i conflitti attraverso la negoziazione.

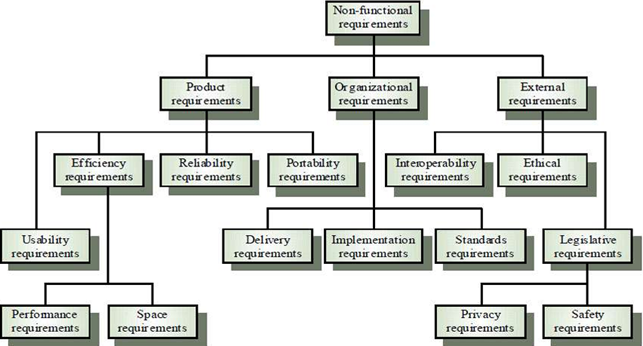
Specifica dei requisiti.  
In questa fase si deve cercare di creare la **specifica dei requisiti**, ovvero bisogna riuscire a **rappresentare** ed **esporre** i **requisiti** identificati, **prioritizzati** e **categorizzati** nelle fasi precedenti.

Esistono varie notazioni per esporre i requisiti:

|  |  |
| --- | --- |
| Notazione | Descrizione |
| Linguaggio Naturale | I requisiti sono scritti usando frasi numerate in lingua naturale. Ogni frase dovrebbe esprimere un requisito. |
| Linguaggio Naturale Strutturato | I requisiti sono scritti in lingua naturale, seguendo uno standard o un template. Ciascun campo fornisce informazioni su un aspetto del requisito |
| Linguaggio Descrittivo Di Progettazione | Questo approccio utilizza il linguaggio come un linguaggio di programmazione, ma con caratteristiche più astratte per specificare i requisiti definendo un modello operativo del sistema. Questo approccio è raramente utilizzato. |
| Notazione Grafica | I modelli grafici, integrati da annotazioni testuali, vengono utilizzati per definire i requisiti funzionali del sistema; i diagrammi di casi d’uso e UML sono comunemente utilizzati. |
| Specifica Matematica | Questa notazione si basa su concetti matematici. Spesso le ambiguità si riducono ma non tutti gli utenti accettano questo tipo di notazione. |

Requisiti non funzionali.  
I **requisiti non funzionali** sono requisiti che non riguardano direttamente i servizi specifici del sistema ma bensì le “**qualità**” che il sistema deve avere.

I requisiti non funzionali indicano le **caratteristiche** o i **limiti** che deve avere il sistema nel suo complesso. Possono quindi riferirsi a proprietà del sistema come l’affidabilità, i tempi di risposta e l’uso della memoria.

Classificazione dei requisiti non funzionali.  
  


Come possiamo vedere dalla figura, ci sono tre categorie principali di requisiti non funzionali:

* **Requisiti di prodotto**: **specificano** o **limitano** il **comportamento** **del** **software**; per esempio, i requisiti prestazionali come la velocità di risposta, la quantità di memoria richiesta, ecc…;
* **Requisiti organizzativi**: **requisiti** **generali** del sistema che derivano dalle **politiche** e dalle **procedure** delle **organizzazione** del cliente e dello sviluppatore; per esempio di standard da seguire, il linguaggio di programmazione, l’ambiente di sviluppo, ecc…;
* **Requisiti esterni**: **requisiti** che derivano da **fattori esterni**; per esempio, i requisiti normativi che stabiliscono cosa deve essere fatto affinché il sistema sia approvato dagli enti di controllo.

Tra i requisiti non funzionali più comuni troviamo:

* Performance (concetto di latenza e produttività);
* Usabilità;
* Affidabilità;

MODELLAZIONE

La modellazione.  
Cos’è un **modello**?

Un **modello** è **un’astrazione del sistema**. Un modello viene utilizzato per **descrivere**, **rappresentare**, **analizzare**, **simulare** e **generare** un sistema software (prima ancora della costruzione vera e propria).

Utilizzare i modelli è molto utile in quanto consente di **catturare** quelli che sono i **caratteri** **principali** del sistema che si vuole sviluppare tralasciando i dettagli inutili.

I modelli inoltre consentono di rappresentare il sistema sotto molteplici punti di vista:

* **Rappresentazione esterna**: viene modellato il contesto o l’ambiente in cui opera il sistema;
* **Prospettiva di interazioni**: vengono modellate le interazioni tra il sistema e il suo ambiente o tra i componenti del sistema;
* **Prospettiva strutturale**: viene modellata l’organizzazione del sistema o la struttura dei dati elaborati;
* **Prospettiva comportamentale**: vengono modellati il comportamento dinamico e le risposte agli eventi.

Attività di modellazione.  
Durante la modellazione di un sistema, al fine di estrarre e semplificare informazioni complesse, possono essere effettuate delle vere e proprie operazioni:

* **Decomposizione**;
* **Gerarchizzazione**;
* **Coupling** e **Cohesion**;

Decomposizione.  
La **decomposizione** è il processo di **suddividere un sistema complesso in parti più piccole e gestibili**. Queste parti sono spesso chiamate “**moduli**” o “**componenti**”, e possono essere trattate separatamente per semplificare il processo di progettazione e sviluppo.

La decomposizione può essere di due tipi:

* **Decomposizione funzionale**: il sistema è decomposto in funzionalità;
* **Decomposizione strutturale**: il sistema è decomposto in classi. Ogni classe rappresenta un’entità;

Gerarchizzazione.  
La **gerarchizzazione** è l’organizza**zione delle parti del sistema in una struttura gerarchica**. Questo concetto esprime come gli oggetti che fanno parte di un sistema si relazionano tra di loro. Questo modello è spesso applicato attraverso:

* **L’ereditarietà**: capacità di definire nuove componenti software basate su componenti esistenti, catturando e riutilizzando le caratteristiche delle componenti esistenti;
* **L’aggregazione**: relazione in cui un componente software è composto da altri componenti in modo che possano essere gestiti come una singola unità.

Entrambe queste due consentono di definire strutture gerarchiche all’interno del sistema.

Coupling e Cohesion.  
Quando si progetto un sistema si cerca di renderlo il più flessibile possibile per rispondere ai cambiamenti nel tempo.

Introduciamo quindi due concetti che ci consento di misurare in maniera informale la flessibilità di un software:

* **Coupling**: **misura la dipendenza tra i sottosistemi**;
* **Cohesion**: **misura la dipendenza tra le classi**;

Un software, in basa alla struttura dei suoi moduli, può avere un:

* **High coupling**: cambiare un sottosistema ha un alto impatto sugli altri sottosistemi;
* **Low Coupling**: cambiare un sottosistema non ha effetto sugli altri sottosistemi;

Un modulo, in base alla propria struttura, può avere una:

* **High Cohesion**: le classi del sottosistema svolgono compiti simili e sono collegate tra loro tramite associazioni;
* **Low Cohesion**: tante classi ausiliari ma con nessuna associazione tra di loro.

La combinazione ideale perché un software possa essere flessibile è:

**Low – Coupling** : **High – Cohesion**

UML

Unified Modeling Language (UML).  
L’**Unified Modeling Language** (**UML**) è il **linguaggio standard** per la **modellazione** di sistemi software.

Più specificatamente UML mette a disposizione dei servizi tramite cui è possibile visualizzare, specificare, costruire e documentare gli elaborati di un sistema software, migliorando la comunicazione tra i vari ruoli all’interno di un processo di sviluppo software.

Tipo di diagrammi.  
UML mette a disposizione vari di diagrammi. Ogni diagramma serve a rappresentare una prospettiva diversa del progetto.

I diagrammi possono appartenere a due macro famiglie:

|  |  |
| --- | --- |
| Diagrammi statici | Digrammi dinamici |
| Rappresentano la struttura base del sistema da realizzare. | Enfatizzano il comportamento del sistema. |

I principali diagrammi che mette a disposizione UML sono:

* **Use Case Diagram;** (dinamico)
* **Component Diagram;** (statico)
* **Class Diagram;** (statico)
* **Sequence Diagram;** (dinamico)
* **State Diagram**; (statico)

USE CASE DIAGRAM

Use Case Diagram.  
I **diagrammi dei casi d’uso** (**Use Case diagrams**) sono uno dei tipi di diagrammi di UML. Essi vengono utilizzati all’interno dell’ingegneria del software per rappresentare le **interazioni** tra un **sistema software** e i suoi **attori**.

Gli use case diagrams forniscono una visione ad alto livello di come un **sistema software è destinato a essere utilizzato**, mostrando in che modo gli **attori** (che possono essere utenti, altri sistemi software o entità esterne) **interagiscono** con il **sistema** per ottenere determinati obiettivi o svolgere specifiche funzioni.

Gli use case diagrams sono composti da:

* **Attori**: gli attori rappresentano le entità esterne al sistema che interagiscono con esso. Questi possono essere utenti umani, altri sistemi software, o qualsiasi entità coinvolta nelle interazioni con il sistema. Gli attori sono rappresentati come degli omini stilizzati.



* **Casi d’uso**: i casi d’uso sono azioni o funzioni specifiche che il sistema software offre agli attori. Rappresentano le diverse modalità in cui gli attori possono interagire con il sistema per ottenere risultati desiderati. I casi d’uso sono rappresentati come ovali all’interno del diagramma;

****

* **Relazioni**: le relazioni tra i casi d’uso e gli attori sono rappresentate da linee o frecce nel diagramma. Queste relazioni possono essere di diversi tipi, tra cui **associazione**, **inclusione** ed **estensione**, e servono ad indicare come gli attori utilizzano i casi d’uso per raggiungere i loro obiettivi.

|  |
| --- |
| **Associazione** |
|  |
| **Descrizione**: rappresenta tipicamente una relazione tra un attore e un caso d’uso. Viene utilizzata per mostrare che un attore può svolgere quell’azione. |

|  |
| --- |
| **Inclusione** |
|  |
| **Descrizione**: rappresenta una relazione tra due casi d’uso. Viene utilizzata per esprimere che un caso d’uso (in questo caso A) può essere svolto solo dopo aver svolto il caso d’uso che viene incluso (in questo caso B). |

|  |
| --- |
| **Estensione** |
|  |
| **Descrizione**: rappresenta una relazione tra due casi d’uso. Viene utilizzata per esprimere che il comportamento di un caso d’uso può essere esteso o ampliato in alcune circostanze. |

COMPONENT DIAGRAM

Progettazione architetturale.  
La **progettazione architetturale** si occupa dell’orga**nizzazione di un sistema software e della progettazione della sua struttura complessiva**.

La fase di progettazione architetturale è la fase intermedia che funge da collegamento tra la progettazione (scrittura di codice) e l’ingegneria dei requisiti, in quanto **identifica le principali componenti strutturali di un sistema e le loro relazioni**.

Component Diagram.  
I **diagrammi dei componenti** (**component diagrams**) sono un tipo di diagramma di UML. Questi tipi di diagrammi ci consentono di rappresentare la struttura di un sistema software.

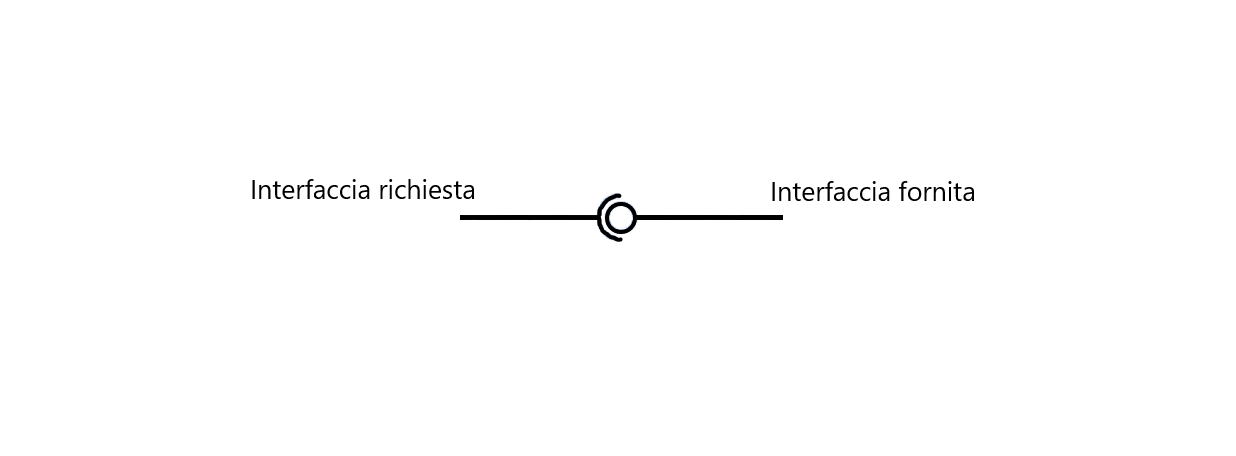
Questo diagramma ci consente di mostrare come le diverse parti o componenti di un sistema si collegano e interagiscono tra di loro.

I component diagrams sono composti da:

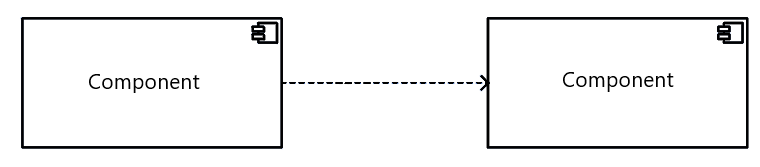
* **Componenti**: un componente è **un’unità di sistema** **autonoma** che all’occorrenza è facilmente riconoscibile e sostituibile. Può rappresentare una parte del codice sorgente, un modulo software, una libreria o anche un componente hardware;

****

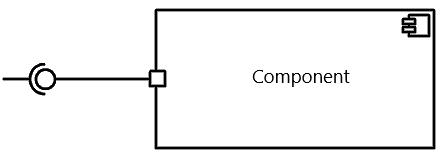
* **Interfacce**: i componenti possono avere interfacce che definiscono come possono essere utilizzati o come comunicano con altri componenti. Queste interfacce possono includere metodi, attributi o altri dettagli utili alla comunicazione tra due componenti;



* **Dipendenze**: una dipendenza è una relazione tra due componenti nella quale un componente chiede all’altra componente un elemento per la sua implementazione completa.



* **Porte:** una porta rappresenta un punto di interazione tra un componente e il suo ambiente. Solitamente quando un componente utilizza una porta sta ad indicare che delegherà le interfacce ad una classe interna. Viene indicata con un quadratino.

****

CLASS DIAGRAM

Class Diagram.  
I **diagramma di classe** (**class diagrams**) sono un tipo di diagramma di UML. Questi tipi di diagrammi ci consentono di **rappresentare la struttura statica di un sistema software**. Essi forniscono un **visione** visuale e **organizzata delle classi** che compongono il sistema e delle relazioni tra di esse.

I class diagrams sono composti da:

* **Classi**: rappresentano le entità chiave del sistema, come oggetti del mondo reale o componenti software;
* **Attributi**: caratteristiche o proprietà delle classi;
* **Metodi**: azioni o operazioni che le classi possono eseguire;
* **Package**: meccanismo per organizzare gli elementi in gruppi;
* **Relazioni**: connessioni tra le classi, come associazioni, ereditarietà e dipendenze;
* **Molteplicità**: definisce quanti oggetti partecipano alla relazione.

|  |  |  |
| --- | --- | --- |
| **Relazione** | **Simbolo** | **Significato** |
| **Associazione** |  | Rappresenta una relazione tra due o più classi, indicando che le istanze di una classe sono associate alle istanze dell’altra. |
| **Generalizzazione** |  | Rappresenta una relazione di ereditarietà tra una classe base e una derivata. |
| **Aggregazione** |  | Rappresenta una relazione “tutto-parte” tra una classe (parte) e un’altra classe (tutto). Indica che l’istanza della parte può esistere anche indipendentemente da tutto il resto. |
| **Composizione** |  | Rappresenta una relazione “parte-di” dove l’istanza della parte è strettamente legata all’istanza del tutto, e che la parte è creata o distrutta insieme al tutto. |

SEQUENCE DIAGRAM

Sequence diagram.  
I **diagrammi di sequenza** (**sequence diagrams**) sono un tipo di diagramma di UML. Questi tipi di diagrammi ci consentono di rappresentare le **interazioni tra gli oggetti all’interno di un sistema e mostrano la sequenza temporale di messaggi scambiati tra di essi durante uno specifico scenario di esecuzione**.

I sequence diagrams sono composti da:

* **Attori**: coloro che svolgono le azione (nel senso pratico sono le classi che effettuano le chiamate dei metodi);
* **Barra di attivazione**: riquadro posto sulla linea di vita (lifeline). Si usa per indicare che un oggetto è attivo durante un’interazione tra due oggetti.
* **Messaggio sincrono** (freccia con punta piena): un messaggio sincrono viene utilizzato quando il mittente attende che il destinatario elabori il messaggio e ritorni prima di continuare con un altro messaggio;
* **Messaggio asincrono** (freccia con punta stilizzata): un messaggio asincrono viene utilizzato quando il mittente non aspetta che il destinatario elabori il messaggio e ritorni prima di inviare altri messaggi ad altri oggetti;
* **Messaggio di ritorno** (freccia tratteggiata): indica che un destinatario ha terminato l’elaborazione e ha restituito un risultato;
* **Distruzione di un oggetto** (una X): indica che l’oggetto ha terminato il suo ciclo di vita e quindi viene distrutto.

Combined Fregment.

Durante gli anni questo tipo di diagramma ha subito numerosi aggiornamenti. Uno dei più importanti è l’introduzione dei **Combined Fragment** (**CF**).

Questo concetto permette di suddividere un sequence diagram in più “**frammenti**” e di “**combinare**” tali frammenti, seguendo diverse regole.

Alcuni operatori, chiamati “**Interaction** **Operator**”, definiscono come tali frammenti possono collegarsi tra di loro.

|  |  |
| --- | --- |
| Operatore | Significato |
| ALT | Rappresenta una scelta. Il frammento viene diviso in più sezione nel quale sono rappresentate le alternative che possono verificarsi. Solo una delle alternative può e deve verificarsi durante l’interazione. |
| OPT | Rappresenta un’opzione. Il frammento conterrà delle istruzioni che possono avvenire come possono non avvenire in base alla circostanza. |
| PAR | Rappresenta il parallelismo. Il frammento viene diviso in più parti che possono essere eseguite in qualsiasi ordine. L’unico vincolo è che l’ordine imposto all’interno di ciascun frammento deve essere rispettato. |
| SEQ | Variazione del PAR. L’ordine interno di ciascun frammento viene rispettato. I frammenti di diverse lifeline possono occorrere in qualsiasi ordine. Eventi in frammenti diversi e stessa lifeline sono ordinati in modo tale da preservare l’ordine. |
| NEG | Rappresenta la negazione. Il frammento viene eseguito solo se una determinata condizione è falsa. |
| CRITICAL | Rappresenta una sezione critica. Le operazioni all’interno del frammento devono essere eseguite in maniera atomica. |
| LOOP | Rappresenta un loop. Le operazioni all’interno del frammento devono essere eseguite fino a quando una determinata condizione è vera. |
| REF | Rappresenta la referenziazione. Possiamo richiamare altri sequence attraverso questo operatore. |

Stato.  
Ad una lifeline può essere associato uno **stato**. Lo stato viene valutato e:

* Se è verificato, allora si può continuate con il proseguimento del sequence diagram;
* Se non è verificato, il sequence diagram non può essere verificato.

Transition time.  
Si possono esprimere **vincoli temporali** sul tempo di esecuzione dei messaggi. Solitamente questi vincoli si posizionano alla destra dei messaggio.

STATE DIAGRAM

State Diagram.  
I **diagrammi di stato** (**state diagrams**) sono un tipo di diagramma di UML. Questi tipi di diagrammi ci consentono di rappresentare la **struttura dinamica di un sistema software**. **In particolare essi modellano il comportamento degli oggetti che possono trovarsi in vari stati durante il loro ciclo di vita**.

Gli state diagrams sono composti da:

* **Stati** (con **eventi** e **attività**): rappresenta una condizione o una situazione specifica in cui un oggetto può trovarsi durante il suo ciclo di vita. Si indica con un rettangolo con gli angoli smussati. È composto da due campi:

|  |  |
| --- | --- |
| Nome | Transazioni interne |
| Una stringa testuale che distingue uno stato da un altro | Lista di attività o azioni interne che sono eseguite mentre l’elemento è all’interno dello stato. Tali azioni non comportano il cambiamento dello stato. |
| Rappresentazione | |
|  |  |

Le transizioni interne hanno la seguente forma:

**label/action**

Dove **label** identifica la circostanza in cui viene eseguita l’azione specifica per quella label. Le label principali sono: entry (l’azione si verifica quando si entra nello stato), exit (l’azione di verifica quando si esce dallo stato), do (l’azione si verifica ripetutamente se lo stato non viene modificato).

* **Transizioni**: è una relazione tra due stati la quel indica che un oggetto nel primo stato entrerà nel secondo stato al verificarsi di un determinato evento, detto “evento scatenante”, quando una condizione viene soddisfatta.

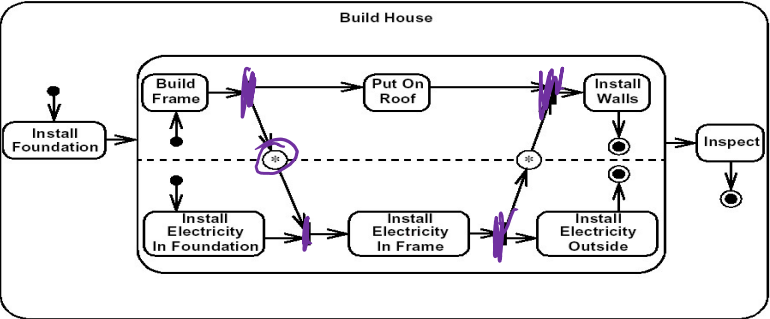
Evento scatenante.  
Verificarsi di un qualcosa che può innescare una transizione di stato. Ci sono quattro tipi di evento:

* **Change event**: si verifica quando il valore di una determinata condizione passa da True a False;
* **Time** **event**: il passare di un determinato lasso di tempo;
* **Segnale**: invio di un evento in modo sincrono;
* **Calls**: ricezione di una chiamata di operazione;

Stati composti.  
Uno stato che ha sotto-stati viene detto **stato composto**.  
Può essere decomposto in:

* **Due o più sotto-stati correnti** (detti **regioni**);
* **Due o più sotto-stati sequenziali**.

Synch States.  
Un **synch state** viene utilizzato per **sincronizzare regioni concorrenti**. Viene utilizzato per assicurare che una regione esca da un particolare stato prima che un’altra regione entri un particolare stato.

Viene rappresentano mediante un cerchio al cui interno vi un “\*”.  
  


Glossario.

Cap. 1 (Ingegneria del software)

* **Ingegneria del software**: disciplina che si occupa dello sviluppo, della manutenzione e della gestione dei software.
* **Software**: documentazione, librerie, siti web, dati di configurazione e codice associati ad un software.

Cap. 2 (Processi di sviluppo)

* **Processo di sviluppo**: insieme di attività che porta alla creazione di un software, e comprende: specifica del progetto, sviluppo del software, convalida dei requisiti e evoluzione del software.
* **Modelli di processi** **software**: rappresentano metodologie organizzative per affrontare lo sviluppo del software.
* **Modello a cascata**: modello che suddivide il processo di sviluppo software in una sequenza lineare di fasi ben definite, ognuna delle quali può iniziare solo se la precedente fase si è conclusa. Le fasi del modello sono: analisi e definizione dei requisiti, progettazione del sistema e del software, implementazione, integrazione e test e rilascio e manutenzione.
* **Sviluppo incrementale (Processi evoluzionari)**: modello di sviluppo che prevede di sviluppare un’implementazione iniziale, esporla agli utenti e perfezionarla attraverso molte revisioni, finché non si ottiene il risultato richiesto.
* **Modello a spirale**: modello caratterizzato da un ciclo di vita a forma di spirale, dove le attività vengono svolte passo dopo passo ma in tempi più brevi, per poi ripatire dall’inizio. Le fasi del modello sono: pianificazione, analisi e rischio, progettazione e sviluppo e valutazione e pianificazione.

Cap 3. (RUP)

* **RUP (Relation Unified Process)**: framework di sviluppo software, ovvero un insieme organizzato e strutturato di linee guida, processi, procedure, modelli e strumenti che forniscono una struttura per la gestione del ciclo di vita di un software.
* **Milestone**: punti di controllo prodotto dopo ogni fase del RUP che permettono di valutare lo stato del processo e la direzione da prendere.
* **Fase iniziale (inception)**: definizione degli obiettivi, identificazione dei rischi, fattibilità, stima dei tempi e dei costi.
* **Fase di elaborazione (elaboration)**: progettazione dettagliata del sistema e in particolare dell’architettura. Si affrontano i rischi principali.
* **Fase di costruzione (costruction)**: fase principale di sviluppo, in cui viene effettivamente scritto il codice.
* **Fase di transizione (transition)**: realizzazione finale e consegna del prodotto finale.
* **Concetti statici**: aspetti che vengono definiti nello stesso modo per tutti i processi. Questi aspetti sono: i ruoli, le attività, gli elaborati e il workflow.
* **Ruoli (chi)**: identificano il comportamento e le responsabilità di un individuo o di un gruppo di individui.
* **Attività (come)**: identificano i compiti specifici, come l’analizzare, l’elaborare o il revisionare, che i membri svolgono durante il processo.
* **Elaborati (cosa)**: rappresentano i risultati tangibili delle attività svolte durante il processo.
* **Workflow (quando)**: rappresenta la sequenza temporale delle attività e degli eventi all’interno del processo di sviluppo.

Cap. 4 (Agile)

* **Sviluppo agile**: insiemi di metodi di sviluppo software emerso come risposta alle sfide riscontrate nello sviluppo del software all’interno di un ambiente in rapida evoluzione.
* **Programmazione estrema**: modello che tende a spingere le normali pratiche a livelli estremi.
* **Scrum**: framework per organizzare progetti agile. Fornisce, in una certa misura, una visibilità esterna su ciò che sta accadendo.
* **Sprint**: arco di tempo in cui viene prodotto parte del software. Alla fine di ogni sprint è prevista la consegna incrementale del software.
* **Ruoli**: figure presenti all’intero di un processo Scrum.
* **Artefatti**: documenti presenti all’interno di un processo Scrum.
* **Eventi**: riunione presente all’interno di un processo Scrum.
* **Product owner**: colui che indentifica le caratteristiche o i requisiti del prodotto.
* **Scrum** **master**: colui che è responsabile del processo. Coordina e mette in condizioni ottimali il team.
* **Team di sviluppo**: responsabili dello sviluppo del software.
* **Product** **backlog**: documento che contiene la lista di tutti i requisiti necessari per la realizzazione del progetto.
* **Sprint backlog**: documento che definisce tutti i task da completare nei singoli sprint.
* **Increments**: tutti i task completati durante un determinato sprint.
* **Sprint** **planning**: riunione in cui il product owner stila product backlog, in presenza del team e dello scrum master.
* **Daily scrum**: riunione tra lo scrum master e il team. Viene annotato il lavoro svolto il giorno precedente per programmare le prossime 24 ore.
* **Sprint Review**: riunione fatta ad ogni fine sprint per verificare le attività svolte per il raggiungimento degli obiettivi in quello sprint.
* **Sprint Retrospective**: riunione volta a migliorare e perfezionare costantemente il processo e la realizzazione del prodotto.

Cap. 5 (Project Planning)

* **Project planning**: processo di definizione, organizzazione e gestione delle attività necessarie per portare a termine con successo un progetto software.
* **Accordo**: punto di partenza di ogni progetto. Comprende la definizione degli obiettivi, requisiti e vincoli che saranno poi formalizzati in un contratto (Process Initiation Document)
* **Pianificazione**: fase che prevede la definizione come il progetto sarà gestito dall’inizio.
* **Azione**: monitoraggio e (eventualmente) riadattamento del progetto.
* **Rivisitazione**: analisi di fine progetto in cui viene osservato ciò che è andato male, ciò che andato bene ciò che può essere migliorato.
* **PERT**: tecnica di gestione dei progetti utilizzata per pianificare le attività.
* **Completion Time**: tempo necessario a sviluppare il progetto.
* **Critical Path**: cammino in cui sono presenti tutte le attività il cui ritardo nel completamento potrebbe causare un ritardo generale nella consegna del progetto.
* **Forward Path**: percorso per analizzare il complition time. Si percorrono le attività dalla prima all’ultima.
* **Backward Path**: percorso per analizzare il critical path. Si porcorrono le attività dell’ultima alla prima.
* **Total Float**: valore che ci aiuta a calcolare il critical path.

Cap. 6 (Ingegneria dei Requisiti)

* **Requisito**: funzionalità o vincolo che un software deve rispettare.
* **Ingegneria dei requisiti**: disciplina che si occupa di scoprire i requisiti di un sistema.
* **Requisiti funzionali**: requisiti che descrivono “ COSA” deve fare un sistema. Descrive le funzionalità che il software deve avere.
* **Requisiti non funzionali**: requisiti che descrivono le qualità che deve avere un sistema software.
* **Requisiti di prodotto**: qualità o vincoli che un sistema deve avere.
* **Requisiti organizzativi**: vincoli che comprendono le politiche e gli standard delle organizzazioni.
* **Requisiti esterni**: requisiti che riguardano tutti i fattori esterni.

Cap. 7 (Modellazione)

* **Modello**: un’astrazione del sistema. Viene utilizzato per descrivere, rappresentare, analizzare, simulare e generare un sistema software.
* **Decomposizione**: suddivisione un sistema complesso in parti più piccole e gestibili.
* **Decomposizione funzionale**: decomposizione di un sistema in base alle funzionalità.
* **Decomposizione strutturale**: decomposizione di un sistema in base alla struttura.
* **Gerarchizzazione**: organizzazione del sistema in una struttura gerarchica.
* **Ereditarietà**: definire nuove componenti software basate su componenti esistenti.
* **Aggregazione**: relazioni in cui un componente software è composto da altri in modo che possano essere gestiti come una singola unità.
* **Coupling**: misura la dipendenza tra i sottosistemi.
* **Cohesion**: misura la dipendenza tra le classi.
* **High Coupling**: cambiare un sottosistema ha un alto impatto sugli altri sottosistemi.
* **Low Coupling**: cambiare un sottosistema non ha effetto sugli altri sottosistemi.
* **High Cohesion**: le classi del sottosistema svolgono compiti simili e sono collegate tra loro tramite associazioni.
* **Low Cohestion**: tante classi ausiliari ma con nessuna associazione tra di loro.

Cap. 8 (UML)

* **UML (Unifed Modeling Language)**: linguaggio standard per la modellazione. Permette di visualizzare, specificare, costruire e documentare gli elaborati di un sistema software.

Cap. 9 (Use Case Diagram)

* **Use Case diagram**: diagramma dinamico. Mostra come il sistema software è destinato a essere utilizzato, mostrando in che modo gli attori interagiscono con il sistema.

Cap. 10 (Component Diagram)

* **Progettazione Architetturale**: si occupa dell’organizzazione di un sistema software e della progettazione della struttura camplessiva.
* **Component Diagram**: diagramma statico. Consentono di rappresentare la struttura di un sistema software.

Cap. 11 (Class Diagram)

* **Class Diagram**: diagramma statico. Consente di rappresentare la struttura statica di un sistema software. Fornisce una visione visuale e organizzata della classi che compongono il sistema e delle relazioni tra di esse.

Cap. 12 (Sequence Diagram)

* **Sequence Diagram**: diagramma dinamico. Rappresenta le interazioni tra gli oggetti all’interno di un sistema e mostrano la sequenza temporale di messaggi scambiati tra di essi durante uno specifico scenario di esecuzione.
* **Combined Fregment**: facoltà di suddividere sequence diagram in più frammenti e combinare tali frammenti.
* **Interaction Operator**: operatori che stabiliscono il comportamento dei frammenti.

Cap. 13 (State Diagram)

* **State Diagram**: diagramma dinamico. Rappresenta il comportamento degli oggetti che possono trovarsi in vari stati durante il loro ciclo di vita.
* **Stato Composto**: uno stato composto da altri sotto-stati.
* **Regione**: uno stato composto da stati organizzati in parallelo.
* **Synch State**: possibilità di due regioni di essere sincronizzate.

Indice.

A.

- **Accordo** ->

- **Aggregazione** ->

- **Artefatti** ->

- **Attività** ->

- **Azione** ->

B.

- **Backword path** ->

C.

- **Class** **diagram** ->

- **Cohesion** ->

- **Combined** **fragment** ->

- **Complition** **time** ->

- **Component** **diagram** ->

- **Concetti** **statici** ->

- **Costruction** ->

- **Coupling** ->

- **Critical** **path** ->

D.

- **Daily scrum** ->

- **Decomposizione** ->

- **Decomposizione** **funzionale** ->

- **Decomposizione strutturale** ->

E.

- **Elaborati** ->

- **Elaboration** ->

- **Ereditarietà** ->

- **Eventi** ->

F.

- **Forward** **path** ->

G.

- **Gerarchizzazione** ->

H.

- **High coupling** ->

- **High cohesion** ->

I.

- **Inception** ->

- **Increments** ->

- **Ingegneria del software** ->

- **Interaction operator** ->

L.

- **Low cohesion** ->

- **Low coupling** ->

M.

- **Milestone** ->

- **Modelli di processi software** ->

- **Modello** ->

- **Modello a cascata** ->

- **Modello a spirale** ->

P.

- **PERT** ->

- **Pianificazione** ->

- **Processo** **software** ->

- **Product backlog** ->

- **Product owner** ->

- **Progettazione architetturale** ->

- **Programmazione estrema** ->

- **Project planning** ->

R.

- **Refactoring** ->

- **Regione** ->

- **Requisiti di prodotto** ->

- **Requisiti esterni** ->

- **Requisiti funzionali** ->

- **Requisiti non funzionali** ->

- **Requisiti organizzativi** ->

- **Requisito** ->

- **Rivisitazione** ->

- **Ruoli** ->

- **RUP** ->

S.

- **Scrum** ->

- **Scrum master** ->

- **Sequence** **diagram** ->

- **Software** ->

- **Sprint** ->

- **Sprint** **backlog** ->

- **Sprint planning** ->

- **Sprint retrospective** ->

- **Sprint review** ->

- **State diagram** ->

- **Stato composto** ->

- **Sviluppo agile** ->

- **Sviluppo incrementale** ->

- **Synch state** ->

T.

- **Team di sviluppo** ->

- **Total float** ->

- **Transaction** ->

U.

- **UML** ->

- **Use case diagram** ->

W.

- **Workflow** ->