
INGEGNERIA E ARCHITETTURA DEL SOFTWARE

Biagi Alessio - Informatica Applicata (2023) - Università di Urbino “Carlo Bo”

CAPITOLO 1: FONDAMENTI DELL'INGEGNERIA DEL SOFTWARE.

1.1. – NOZIONI FONDAMENTALI

Lo studio dell'Ingegneria ed Architettura del Software **nasce nel 1940**, con la nascita dei primi computer nelle università e nei centri di ricerca militare, che erano grandi come stanze.

Gli utenti di queste macchine erano fisici, matematici o ingegneri elettronici, dato che il **loro uso era complesso**, dato anche il senso della programmazione che era manuale come un “arte”.

Successivamente, negli **anni 1950**, questi macchinari **ridussero il loro spazio** da stanze ad armadi.

Si iniziò ad **utilizzarli anche nel campo commerciale**, permettendo così **la nascita dei primi linguaggi di basso livello**, permettendo così la creazione della figura del **programmatore** (il quale prima era anche manutentore), nonostante **l'approccio** fosse ancora **“casereccio”**.

Negli **anni 1960**, vi è una **rapida evoluzione della tecnologia hardware**, diventando **sempre più complessa**, dato che codesti sistemi diventano sempre più presenti nell'uso commerciale, causando un **aumento di utilizzatori** (anche non esperti), che porta così a nuove **necessità ed a risoluzione di codeste che dovevano essere accessibili a tutti**.

Tra gli anni **1960/1970**, vi è la **CRISI DEL SOFTWARE**, ovvero l'aumento degli **utilizzatori dei computer e poche persone sapevano programmare un buon software**.

Questa crisi venne evidenziata da codesti aspetti, su molti progetti:

- Andarono fuori dal budget che si era stanziato inizialmente;
- Andarono fuori tempo rispetto alla scadenza;
- Erano di bassi qualità, dato che ancora non esisteva una figura specializzata nella programmazione;
- Non raggiunsero i requisiti dell'utente, rimanendo così invenduti;
- Il codice, data sempre la sua maggiore complessità, diventa sempre più difficile da gestire e mantenere.

La risoluzione è stata sempre più chiara col passare con gli anni, dato che ci sono diversi aspetti che bisogna considerare, e non solo l'implementazione del software, come:

- **Fare una stima del costo del progetto per la sua realizzazione:**
 - Quindi si richiede di stimare il codice che si ottiene alla fine.
- **Suddivisione dei compiti tra i vari membri del team;**
- **Comunicazione fra gli sviluppatori membri del team:**
 - Ovvero, la loro porzione di lavoro deve essere comprensibile agli altri membri del team.
- **Comunicazione con il cliente:**
 - Per permettere così, più facilmente, di soddisfare la richiesta del cliente ed evitare fraintendimenti o altro.
- **Considerare l'evoluzione dei requisiti:**
 - I quali non devono variare troppo, dato che richiederebbe un lavoro infinito.

Questi aspetti sono considerabili come “Aspetti di ingegneria gestionale”.

- **NASCITA DELL'INGEGNERIA DEL SOFTWARE:**

La nascita di codesta scienza, considerata non esatta dato che si basa molto sulla comunicazione, può essere datata negli anni 1968 - 1969, quando il comitato scientifico della NATO ha sponsorizzato due conferenze:

1. In Garmisch, in Germania.
2. A Roma, in Italia.

In **entrambe queste conferenze**, i differenti approcci alla “Ingegneria” usata per costruire un grande software, **concordavano su determinati fattori**:

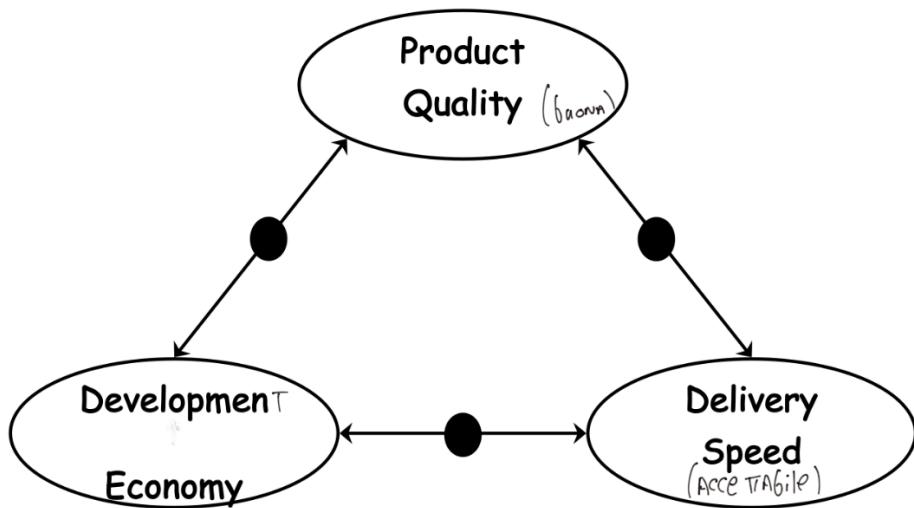
- Management;
- Organizzazione;
- Teorie e principi;
- Metodologie e tecniche;
- Gli strumenti.

Di tale scienza, **si sono avute varie definizioni**:

- 1969 - Riferimento all'uso di principi di “sound engineering” per ottenere un software economicamente sostenibile, efficiente ed affidabile.
- 1990 - Riferimento all'applicazione sistematica, disciplinata e quantificata nello sviluppo, nelle operazioni e mantenimento del software.
- 2006 - Riferimento alla professione delle persone che creano e/o mantengono software, applicando tecnologie pratiche della scienza dell'informatica e altri elementi (come l'uso del project management, applicazioni di domini...).

- TRIADE DELL'INGEGNERIA DEL SOFTWARE:

In questa triade, gli ingegneri non cercano di creare Software perfetti, ma stanno **cercando di creare un “equilibrio” tra il campo economico, qualitativo e tempo**, per permettere di ottenere la massima soddisfazione.



- “IMMATURITÀ” DELL'INGEGNERIA DEL SOFTWARE:

Essendo una scienza relativamente recente, questo causa la “immaturità” dell’industria, creando una situazione dove:

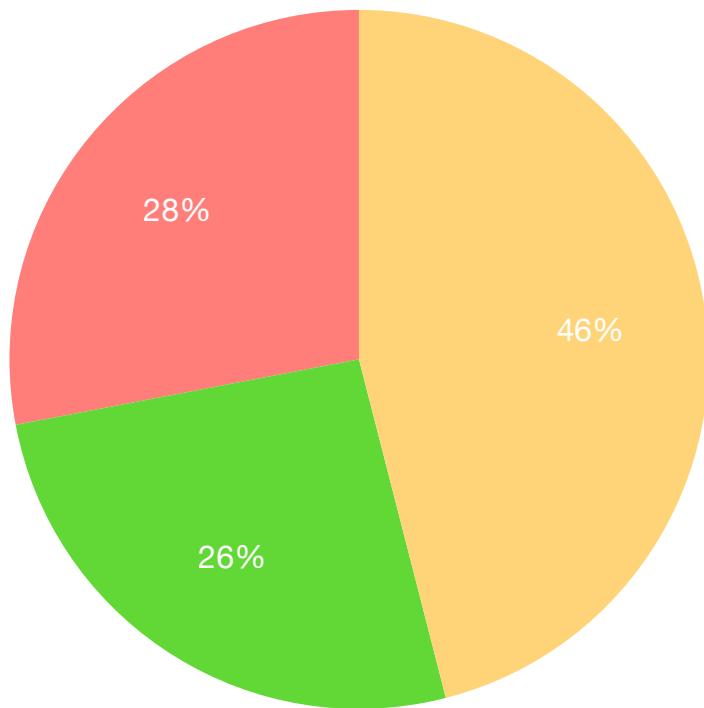
- Il tasso di crescita del mercato ICT è molto alto;
- La complessità dei prodotti software è in aumento;
- Mancanza di competenze tecniche (skills) e persone che non sono sempre formate sulle ultime che vengono richieste.
- Alta percentuale di progetti software che falliscono (come avvenuto con Google+).

*ICT = *Information & Communication Technology*

- FALLIMENTO DEI PROGETTI SOFTWARE:

Possiamo analizzare le percentuali di fallimento, attraverso la ricerca pubblica da Standish Group, che si basa su 28.000 progetti:

- Progetti che erano in ritardo, costi aggiuntivi, funzionalità inadeguate o insufficienti
- Progetti terminati con successo
- Progetti falliti



Quindi abbiamo che:

- La maggior parte dei progetti (46%) ha avuto delle difficoltà di tempo / costi aggiuntivi / le funzionalità erano insufficienti e/o inadeguate.
- Una parte intermedia (28%) hanno fallito.
- La minoranza (26%) hanno terminato il progetto con successo.

Le cause di fallimento:

1. I project manager non comprendevano le esigenze degli utenti;
2. Lo scopo del progetto non era ben definito;
3. Le modifiche al progetto sono state gestite male;
4. La tecnologia richiesta cambia;
5. Le esigenze aziendali richiedono un cambiamento;
6. Le scadenze non sono realistiche;
7. Gli utenti sono “resistenti”;
8. La sponsorizzazione viene persa;
9. Il progetto manca di persone con competenze adeguate;
10. I manager ignorano le buone norme e le lezioni apprese.

Le cause di successo:

1. Requisiti stabili;
2. Stime accurate;
3. Lavoro in gruppo e visione unificata;
4. Attenzione ai rischi.

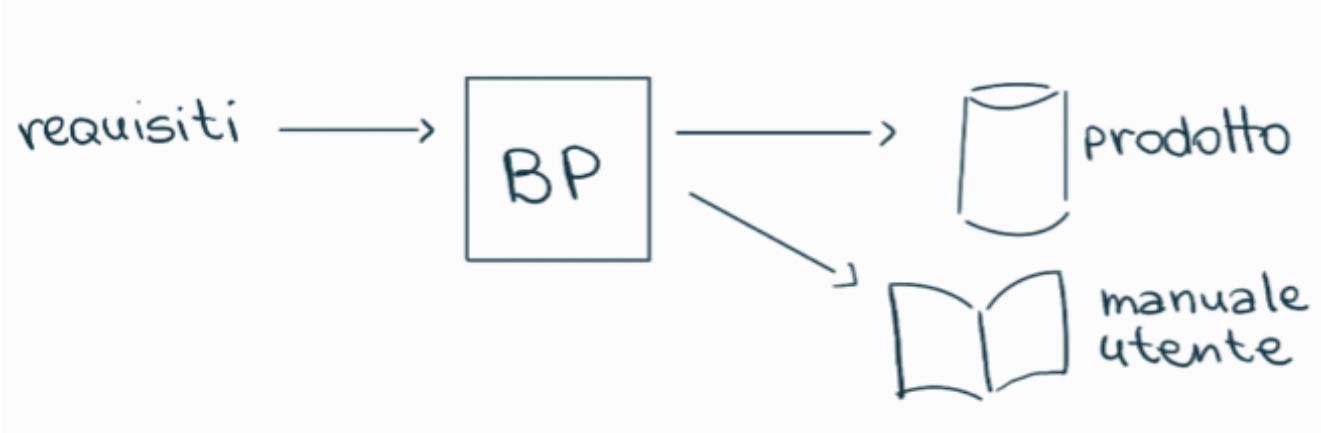
- I PRINCIPI DELL'INGEGNERIA DEL SOFTWARE:

L'ingegneria del Software si occupa dei processi di produzione, principi di sviluppo, tecniche e notazioni (queste ultime per permettere anche a grandi team di poter commentare il codice in modo chiaro per gli altri membri).

L'obiettivo finale è quello di **ottenere un elaborato** che sia di **qualità, sviluppato nei tempi e costi stabiliti** e che **soddisfi** il cliente.

Tutti gli aspetti dell'Ingegneria condividono i seguenti principi:

- Rigore e formalità (ovvero codificare con una modalità specifica);
- Separazione dei problemi;
- Modularità;
- Astrazione;
- Principio di adattamento;
- Generalità;
- Incrementalità.



Dato che i software sono elementi complessi da sviluppare, **serve creare una disciplina rigorosa che la regolamenti, per rendere il codice non ambiguo e chiaro** a tutti, per questo:

1. Ogni membro del team ha un ruolo specifico e tutti i membri devono riuscire a capire la stessa cosa leggendo il codice prodotto.
2. Bisogna considerare che ogni prodotto finale ha un lungo ciclo di vita che va mantenuto, attraverso testing e verifiche di bug.
3. Considerare la necessità di riutilizzare vecchie tecnologie per ridurre i costi di creazione.
4. Il prodotto deve essere mantenuto e considerare sviluppi di esso.

Questi principi esprimono che lo sviluppo della programmazione ad oggetti è una buona norma nel campo dello sviluppo del Software.

- PROCESSO DELLO SVILUPPO DEL SOFTWARE:

Un processo di sviluppo software, detto anche ciclo di vita del software, **è un processo strutturato da sottoprocessi / attività**, che non per forza sono sequenziali (infatti, possono essere eseguiti in parallelo), che sono:

ANALISI DEI REQUISITI E DELLE SPECIFICHE:

Il primo punto dei modelli di sviluppo, dato che è la fase in cui bisognerà comprendere i requisiti e la specifica del prodotto software desiderato.

Questa fase, come ingegneri del software, ci permette di comprendere le necessità del software, anche riconoscendo i requisiti che sono incompleti, ambigui e contraddittori. Oltre a ciò, la specifica è molto importante per le interfacce esterne, che devono essere stabili.

STUDIO DELLA FATTIBILITÀ:

Codesto studio viene eseguito prima del lavoro, dato che ci permette di accertarci della probabilità di successo del progetto, permettendoci di stimare il costo dei componenti.

PROJECT PLANNER:

In questa fase, si inizia a creare il progetto, tramite la suddivisione delle Task principali in Sotto-Task e calendirizzare e assegnare tutte le attività al team.

IMPLEMENTAZIONE / CODING:

Codesta fase prevede la riduzione del progetto in un codice.

Può sembrare la fase più ovvia, ma non necessariamente la più grande.

TESTING:

Prevede "prove" su parti di codice, che devono essere fatte in maniera massiccia, specialmente su parti dove hanno lavorato molti ingegneri insieme, ciò per evitare di mettere prodotti fallati sul mercato e far perdere credibilità nel mercato.

DOCUMENTAZIONE:

Codesta fase è importante, ma spesso trascurata.

Prevede la formazione di una documentazione che mostra la progettazione interna del software, ciò ai fini di manutenzione o miglioramento futuro.

Inoltre, è importante per le interfacce esterne.

VERIFICA DELLA QUALITÀ:

Si esegue una verifica che il codice e la documentazione sia di ottima qualità, prima di consegnarla, per poi far validare ciò direttamente dal cliente.

CONSEGNA DEL PRODOTTO:

La prima versione o una successiva del prodotto viene consegnata al cliente.

MANTENIMENTO DEL PRODOTTO

In questa fase si garantisce il mantenimento del prodotto.

- L'Obiettivo finale:

Gli obiettivi, quindi, della Ingegneria del software è quello di

creare un software che:

- Soddisfi il cliente;
- Che sia adeguatamente affidabile;
- Che sia semplice da capirne il funzionamento e nella sua modifica;
- Che sfrutti componenti e sottosistemi disponibili o appropriati;
- Che abbia una buona efficienza nel suo uso;
- Che i suoi obiettivi rispettino i vincoli di sistema e dell'ambiente;
- Che sia stato realizzato con il miglior compromesso tra risorse disponibili, i costi di realizzazioni, tempi e altro.

- **Principali rami dell'ingegneria del software:**
- **Studio dei processi** che avvengono nello sviluppo di un software;
- **Analisi degli aspetti economici** per la produzione di software;
- **Modellazione** del software;
- **Analisi del software** e metodi di design;
- **Sviluppo di tools e support** per la realizzazione di un software;
- **Analisi di tecniche di testing**, per capire quali siano adeguati per un software o no;
- **Metriche sul software**, ovvero creare misure di produttività o performance del codice.
- **Relazioni** tra le persone coinvolte nel processo di sviluppo del software.

- **IL RUOLO PROFESSIONALE DELL'INGEGNERE SOFTWARE**

Tale figura si è sviluppata nel tempo, per esserlo non bisogna essere solamente un programmatore, ovvero:

“Una figura solitaria che ha molte abilità tecniche in vari ambienti di sviluppo o linguaggi e che vada a creare applicazioni da una specifica, attraverso la programmazione “In The Small””.

Ma per essere codesta figura, **deve**:

- **Creare componenti** che devono essere integrati con altri componenti, per permettere la creazione di un prodotto;
- **Progetta componenti** che possono essere **riutilizzati** in altri sistemi;
- **Progetta ad un livello alto di estrazione**, spesso orientato alla programmazione ad oggetti;
- **Considerare di operare in un team** dove ogni persona partecipa alle diverse fasi dello sviluppo dei processi;
- **Conoscere tutte le fasi e problematiche** di un ciclo del software;
- **Essere capace di prendere decisioni** giuste e / o adeguate in base alla fase in questione.
- Avere un **elevato livello di comunicazione interpersonale**.
- Saper programmare “**In the Large**”.

- MITI SUI SOFTWARE:

“Una dichiarazione generale di obiettivi è sufficiente per iniziare a scrivere programmi, potremmo aggiungere dettagli in seguito”.

La realtà di ciò è che una dichiarazione ambigua sugli obiettivi è la previsione di un disastro, dato che degli obiettivi chiari si sviluppano solo tramite una comunicazione continua tra cliente e sviluppatore.

“I requisiti del progetto cambiano continuamente, ma il cambiamento può essere facilmente adattato dato che il software è flessibile”.

In realtà, se le modifiche sono richieste in anticipo, l'impatto di tale operazione è veramente minimo, ma se codeste vengono richieste sempre più successivamente, l'impatto aumenta rapidamente, dato che bisognerà riconsiderare le stime delle risorse.

“Se rimaniamo in ritardo sotto la tabella di marcia, la risoluzione sarà quella di aggiungere altri programmatori al lavoro e recuperare il ritardo”.
(Concetto di Onda Mongola)

In verità, non essendo un processo meccanico quello della creazione e produzione del software, man mano che arriveranno nuove persone, bisognerà spendere tempo per i nuovi arrivati, togliendone a coloro che già lavoravano.

“Una volta scritto il programma che funziona, il lavoro è finito”.

Assolutamente no, dato che il lavoro rimane per il mantenimento del lavoro realizzato.

“L'unico lavoro da fornire per il successo della richiesta, è il programma”.

Un programma funzionante è solo una parte della configurazione software, la quale rinchiede anche altri elementi, come la documentazione o i commenti nel codice.

CAPITOLO 2: LA QUALITÀ DEL SOFTWARE

CHE COS'È?

Secondo l'ISO (International Organization for Standardization) 8402, è la somma della qualità delle caratteristiche / particolarità che un prodotto possiede per portare avanti l'abilità volta nella soddisfazione di bisogni esplici o impliciti.

! La valutazione nella applicazione di un software, è complicata dato che fa riferimento alla funzione del soggetto, l'analisi del contesto e altri punti di vista.

Quando si parla di qualità, si fa riferimento alla:

- Qualità del processo produttivo: riferito alla “catena di montaggio”, dove se la quale non sarà qualitativa, non potremmo ottenere un prodotto di alta qualità;
- Qualità del prodotto:
 - Esterne;
 - Interne.

E oltre a ciò, nell'uso di un Software, abbiamo diversi punti di vista:

| Utente | Developer | Manager |
|--|--|--|
| Che potrà apprezzare le qualità esterne , come l'interfaccia. | Che apprezzerà soprattutto le qualità interne , come il codice. | Apprezzerà il fatto che sia stato sviluppato con minor budget e tempo possibile. |

Oltre a tutto ciò, la valutazione di un software, è considerata complicata, dato che:

- Il mondo dell'informazione si evolve velocemente:
 - Rendendo così, i software sempre più complicati;
- Molte volte, la qualità del software, è soggetto ad elementi esterni, come: Hardware di sistema / sistema operativo della macchina / connessione di rete, che non si possono prevedere in anticipo, ma che andranno ad attaccarne l'utilizzo.
- L'apprezzamento delle qualità è soggettivo;
- La valutazione dipende dalla conoscenza del progetto.

2.1 QUALITÀ DEL PROCESSO PRODUTTIVO

Gli elementi di valutazione standard sono nati anche prime dell'ingegneria, come:

- ISO 9000: regola la certificazione della qualità del software;
- ISO 9126: definisce il modello di requisiti di qualità di un prodotto software;

Presenti altri che vengono poi rivisti in questi.

Passiamo a parlare della famiglia:

ISO 9000

Famiglia standard che si basa sul presupposto che se si esegue un modello di produzione di qualità, allora si otterrà in automatico un prodotto di ottima qualità.

Sono presenti varie famiglie, dove ognuna fa riferimento ad una microarea di produzione, come:

- ISO **9001**: Si applica alle organizzazioni che sono impegnate alla progettazione / sviluppo / produzione e assistenza di beni.
- ISO **9002**: Si applica alle organizzazioni che non progettano i prodotti ma che sono coinvolte solo nella progettazione (non vengono incluse le organizzazioni di sviluppo software).
- ISO **9003**: Si applica alle organizzazioni che installano o testano i prodotti, come i fornitori del gas.

Quindi, possiamo dire che la norma 9000 con le sue famiglie, definisce una serie di linee guida per il processo produttivo, ma non è direttamente interessato al prodotto stesso.

Successivamente, citiamo il **SEI-CMM** che è uno dei primi modelli che permette di certificare un processo produttivo del software.

2.2 QUALITÀ DEL SOFTWARE

Successivamente della norma ISO 9000, si è creato lo standard per la qualità del prodotto, ovvero l'ISO 9126 che definisce quali standard deve possedere un software.

L'ISO 9126 è divisa in 4 parti che si occupa dei seguenti argomenti:

- **Qualità del modello;**
- **Metriche esterne;**
- **Metriche interne;**
- **Metriche di qualità in uso.**

Queste si misurano attraverso esami e test tramite la compilazione di feedback con il cliente.

Le qualità che deve possedere un software sono le seguenti:

- **Funzionalità**: Grado in cui il software soddisfa le esigenze dichiarate, come indicato dai seguenti sotto-attributi;
 - **Adeguatezza**: Attributi che determinano la presenza / adeguatezza di un insieme di funzioni per compiti specifici;
 - **Accuratezza**: Attributi che influiscono il grado con cui soddisfa il compito.
 - **Interoperabilità**: Attributi del software che influiscono sulla sua capacità di agire con sistemi specifici;
 - **Aderenza delle regolamentazioni**;
 - **Sicurezza**: Attributi del software che influiscono sulla capacità di impedire l'accesso non autorizzato a programmi e dati.
- **Affidabilità**: Quantità di tempo in cui il software è disponibile per l'uso, come indicato dai seguenti sotto-attributi:
 - **Maturabilità**: Grado di capacità del software di gestire gli errori;
 - **Tolleranza agli errori**: In presenza di errori, il grado di prestazione del software degrada e tale attributo identifica il suo grado di tolleranza.
 - **Ripresa del software**: Nel caso di crash del software, grado con il quale riesce a ripristinare la sessione successiva.
- **Usabilità**: Grado di facilità d'uso del software, come indicato dai sotto-attributi:
 - **Comprensibilità**: Attributi del software che influiscono sullo sforzo degli utenti per la comprensione del suo funzionamento.
 - **Apprendibilità**: Attributi del software che influiscono sullo sforzo degli utenti per l'apprendimento del software.
 - **Operabilità**: Attributi che influiscono sullo sforzo degli utenti per il funzionamento e controllo del funzionamento del software.

- **Efficienza**: Grado con cui il software fa uso ottimale delle risorse di sistema, come:
 - **In quanto tempo** restituisce il risultato desiderato;
 - **Quante risorse consuma** per la elaborazione del risultato (come RAM, disco e altro), quindi più consuma e meno sarà qualitativo.
- **Manutenibilità**: Grado di facilità con cui è possibile supportare il software con aggiornamenti o riparazioni, come indicato dai seguenti sotto-attributi:
 - **Analizzabilità**: Sforzo per eseguire a fare analisi per individuare il bug da correggere.
 - **Mutuovolezza**: Sforzo necessario per le modifiche, per essere adattato a un Sistema operativo differente oppure per la correzione di bugs.
 - **Stabilità**: Attributi della capacità software di reagire a modifiche impreviste causate dalle modifiche avvenute.
 - **Testabilità**: Sforzo necessario per validare le modifiche eseguite su un software.
- **Portabilità**: Grado di facilità con cui il software può essere trasportato da un ambiente all'altro, come indicato da:
 - **Adattabilità**: Attributi del software che comportano l'adattamento del software ai diversi ambienti specificati, senza applicare altre azioni o mezzi oltre a quelli previsti a questo scopo per il software desiderato.
 - **Installabilità**: Attributi che indicano lo sforzo necessario per installare il software in un ambiente specifico.
 - **Conformità**: Attributi del software che fanno sì che il software aderisca a standard o convenzioni relative alla portabilità.
 - **Rimpiazzabilità**: Attributi che riguardano l'opportunità e lo sforzo di usare un software al posto di un'altro software specificato nell'ambiente di quel software.



Successivamente, parliamo:

L'**ISO 9126-2/3** fa riferimento alle metriche interne (quelle riferite al codice) ed esterne (vediamo il software come una “scatola nera”, quindi internamente non lo concepiamo, ma lo desumiamo esternamente).

2.3 QUALITÀ DELLE METRICHE IN USO

Dato che le qualità esterne impattano il software, questa qualità saranno disponibili quando il prodotto verrà usato in condizioni reali:

Questo standard si basa su:

- **Fattori;**
- **Criteri;**
- **Metriche.**

L'ISO 9126 distingue due tipologie di errori del programma:

- **Difetto:** Non viene soddisfatto un requisito oppure fallimento nell'incontrare l'utilizzo normale dei requisiti.
- **Non conformità:** Non viene soddisfatto un requisito specifico, causando un fallimento nell'incontrare uno stato specifico, le cause di tale tipologia:
 - Povera comunicazione;
 - Scarsa documentazione;
 - Personale non abbastanza addestrato;
 - Poca motivazione del personale;
 - Pochezza di materiali o strumenti utilizzati.

SQUARE

Dal 1 marzo 2011 la norma ISO/IEC 9126 è stata sostituita dalla norma ISO/IEC 25010:2011 - Ingegneria dei sistemi e del software - Qualità dei sistemi e del software Requisiti e Valutazione (SQuaRE) - Qualità del sistema e del software Modelli.



Rispetto alla 9126 sono state aggiunte "sicurezza" e "compatibilità".

come caratteristiche principali, oltre a ciò sono state rinominate:

- **Funzionalità** → Adeguatezza funzionale;
- **Efficienza** → Efficienza delle performance.

La **sicurezza**, come detto, è una nuova caratteristica che contiene al suo interno:

- **Confidenzialità**: i dati saranno accessibili solo da chi è autorizzato;
- **Integrità**: i dati dovranno essere prodotti da chi non è autorizzato alla modifica;
- **Non-ripudiabilità**: le azioni eseguite devono essere dimostrate che sono esistite;
- **Responsabilità**: le azioni devono poter essere riconducibili a chi le ha eseguite;
- **Autenticità**: l'identità è quella rivendicata.

Mentre, la **compatibilità** contiene:

- La coesistenza viene spostata dalla portabilità;
- Interoperabilità viene spostata dalla funzionalità.

2.4 QUALITÀ DEL SOFTWARE ORIENTATA AGLI OGGETTI

L'ingegneria del software "si sposa" tanto con la programmazione ad oggetti, specialmente per codeste caratteristiche:

- **Estendibilità**:

Facilità con cui un componente facilità l'utilizzo selettivo dei componenti.

- **Riutilizzabilità**:

Grado con cui un sistema / componente facilità l'utilizzo selettivo dei componenti.

- **Modularità**:

Facilità con cui un software può venir trattato come una parte autonoma, come indicato da codesti sotto-attributi:

1. ***Scomponibilità***: facilità con cui un problema può essere scomposto in sottoproblemi.
2. ***Componibilità***: Attributi dei software che favoriscono la produzione di elementi software che possono essere poi liberamente combinati tra di loro per produrre nuovi sistemi.
3. ***Incomprensibilità***: Capacità nel capire come è scomposto il codice e cosa fa ogni sottomodulo.
4. ***Continuità***: Capacità con cui un software si può modificare;
5. ***Protezione***: Capacità di confinare una condizione anomala ad un modulo o, al massimo, a quelli adiacenti.

3. PROCESSO DI SVILUPPO DEL SOFTWARE

La creazione di un software è un processo di apprendimento interattivo, dato dal risultato della conoscenza raccolta e organizzata durante lo svolgimento del processo.

Quindi tale processo di sviluppo del software è un Framework* per le attività richieste per creare software di qualità.

*Framework: piattaforma che funge da strato intermedio tra un sistema operativo e il software che lo utilizza.

Abbiamo diverse tipologie di programmazione:

A. In The Small:

- Composto da un programmatore, dove è presente un modulo composto dalle fasi di modifica - compilazione - debug.

B. In The Large:

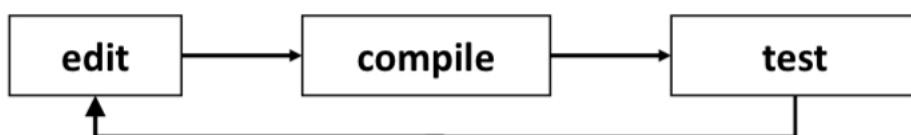
- Prevede di costruire un software composto da più moduli, con diverse versioni e configurazioni.

C. In The Many:

- Prevede la costruzione di un software molto complesso e grande che per la risoluzione prevede la cooperazione e coordinamento con diversi sviluppatori.

IN THE SMALL - PROGRAMMAZIONE E DEBUG:

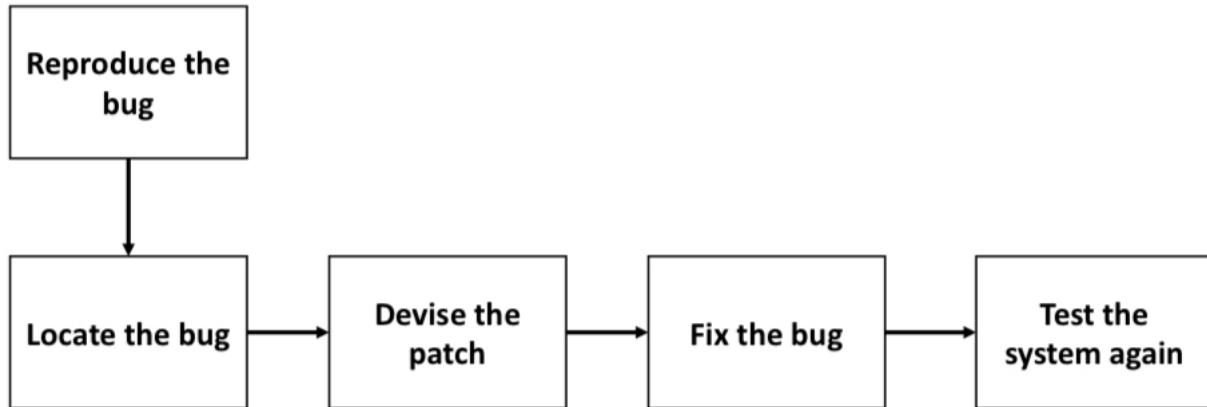
Dove la programmazione è individuale, abbiamo tale schema che evidenzia i passaggi:



Dove avremmo:

| Vantaggi | Svantaggi |
|---|---|
| Feedback molto veloci | Ci si concentra specialmente sulla codifica |
| Diversi strumenti per la creazione software disponibili (IDE) | Nessun incentivo alla creazione della documentazione. |
| | Nessuno stimolo al ridimensionamento nella creazione "In The Large" o "In The Many" |
| | La fase di debugging richiederà un'attività separata |

3.1 IL PROCESSO DI DEBUGGING:



Gli sviluppatori dedicano il 50% del loro tempo di programmazione alla ricerca e alla correzione dei bug.

Secondo uno studio condotto dall'Università di Cambridge, il costo globale del debug stimato è pari a 312 miliardi di dollari all'anno.

Ovviamente, codesto costo è destinato ad aumentare in proporzione alla complessità del software prodotto.

L'università, inoltre ha individuato che pochissima ricerca si concentra sulla fase del debugging.

3.2 FRAMEWORK ACTIVITIES

Tale attività è costituita da uno / più attività che garantiscono l'avanzamento del processo produttivo. Queste attività sono strettamente legate alla realizzazione del processo software e sono generalmente:

- **Comunicazione;**
- **Pianificazione;**
- **Modellazione**, costituita da:
 - Analisi dei requisiti;
 - Progetto;
- **Costruzione**, costituita da:
 - Generazione del codice;
 - Test del codice;
- **Distribuzione.**

Un quadro di processo di queste attività comune è il seguente:

Process framework



UMBRELLA ACTIVITIES:

Queste sono attività ausiliare che non sono strettamente legate alle fasi produttive del processo, ma aiutano al raggiungimento dell'obiettivo.

Generalmente, codeste, sono:

- Tracciamento e controllo del progetto software;
- Gestione del rischio;
- Garanzia di qualità software (SQA: Software Quality Assurance);
- Revisione sulle tecniche formali;
- Misurazione (Measurement);
- Gestione della configurazione del software (SCM: Software Configuration Management);
- Gestione della riusabilità;
- Preparazione e produzione del prodotto di lavoro.

3.3 CMM – CAPABILITY MATURITY MODEL

Questo modello permette di misurare diversi livelli per il progresso e maturità delle organizzazioni software.

Misura la solidità dei processi, ma non la correttezza tecnica o architettura di un prodotto.

Il modello è stato proposto nel 1987 dal “Software Engineering Institute (SEI)”, successivamente nel 2000 è stata rivisitato creando “Capability Maturity Model Integration (CMMI)”.

Questo ultimo modello è composto da cinque livelli di valutazioni e ad ognuno sono associati diversi “Key Process Area (KPA)”, i quali per proseguire con i livelli devono essere tutti soddisfatti.

1° LIVELLO - INIZIALE

Codesto è il livello iniziale, dove il KPA non ha nessuna area di processo chiave ed esprime che:

“L’organizzazione in genere non fornisce un ambiente stabile per lo sviluppo e la manutenzione del software”. Per questo, esistono pochi processi del software che sono stabili e le prestazioni possono essere stabilite solamente in base alle capacità individuali, piuttosto che a quelle organizzative”.

2° LIVELLO - GESTIONE

In questo livello, vengono stabilite le politiche per la gestione di un processo software e le procedure per implementare tali politiche.

La pianificazione e gestione di nuovi progetti si basa sulla esperienza di progetti simili.

Abbiamo codesti KPA:

- Gestione della configurazione del software, che si concentra sulla gestione e sul controllo delle modifiche apportate al software durante il suo ciclo di vita
- Garanzia della qualità del software, che si riferisce a un insieme di pratiche e attività volte a garantire che il software sviluppato soddisfi i requisiti specifici e gli standard di qualità.
- Gestione subappalti software, che si concentra principalmente sulle pratiche di gestione interna per lo sviluppo e la manutenzione del software all'interno di un'organizzazione.
- Monitoraggio e supervisione del progetto software, che si concentra sul monitoraggio e la supervisione dell'avanzamento del progetto software per identificare e risolvere eventuali problemi in modo tempestivo.
- Pianificazione del progetto software;
- Gestione dei requisiti, che si concentra su tutte le attività associate alla definizione, gestione, tracciabilità e verifica dei requisiti del software.

3° LIVELLO - DEFINIZIONE

In questo livello:

Il processo standard per lo sviluppo e la manutenzione del software in tutta l'organizzazione deve essere documentato, compresi anche i processi di ingegneria e gestione del software.

Abbiamo i seguenti KPA:

- Revisione tra pari, permettendo così di individuare e correggere i difetti nei prodotti di lavoro software;
- Coordinamento intergruppo;
- Ingegneria del prodotto software, non si riferisce a un'area specifica del modello, bensì all'ambito generale a cui si applica il CMMI stesso
- Gestione integrata del software, che si concentra sulla gestione integrata e coordinata di tutti i processi di sviluppo software all'interno di un'organizzazione.
- Programmi di formazione per il miglioramento delle competenze e conoscenze dei membri.
- Definizione del processo organizzativo, che identifica e documenta i processi standard e le linee guida operative dell'organizzazione.
- Focus sul processo organizzativo, che contribuisce all'efficacia del Training Program.

4° LIVELLO - GESTIONE DELLA QUANTITÀ DI LAVORO

In questo livello:

L'organizzazione stabilisce obiettivi quantitativi di qualità sia per i prodotti che per i processi software.

La produttività e la qualità vengono misurate per attività importanti dei processi software di tutti i progetti per verificare se le misure sono conformi a ciò che stabilito.

I KPA sono i seguenti:

- Gestione della qualità del software, che si intende un processo sistematico volto a garantire che il software soddisfi i più elevati standard in termini di funzionalità, affidabilità e soddisfazione degli utenti.
- Gestione quantitativa del processo, che si concentra sull'utilizzo di dati quantitativi per migliorare i processi software.

5° LIVELLO - LIVELLO DI OTTIMIZZAZIONE

In questo livello:

Tutta l'organizzazione è focalizzata sul miglioramento continuo dei processi. L'organizzazione ha i mezzi per identificare i punti deboli e migliorare il processo, con l'obiettivo di prevenire il verificarsi di errori o altro.

In fine, i KPA sono:

- Gestione del cambiamento dei processi, che si concentra sulla capacità di un'organizzazione di gestire e controllare le modifiche ai propri processi software in modo efficace e sistematico.
- Gestione del cambiamento tecnologico, che si concentra sulla capacità di un'organizzazione di gestire e controllare l'introduzione di nuove tecnologie nei propri processi software in modo efficace e sistematico.
- Prevenzione dei difetti, che si concentra sulla capacità di un'organizzazione di prevenire la comparsa di difetti nei propri prodotti software.

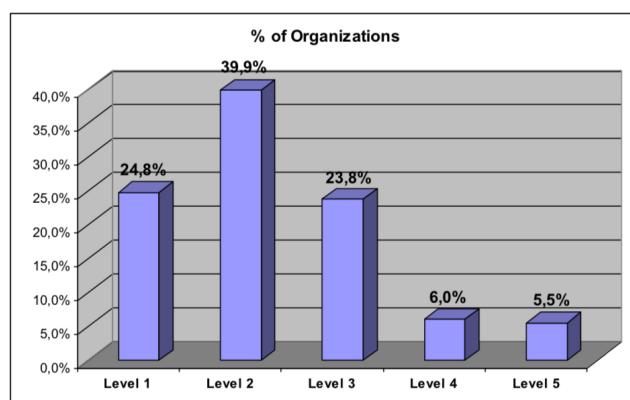
COME RAGGIUNGERE I KPA?

Per il raggiungimento di un KPA sono presenti delle pratiche che sono:

- Obiettivi;
- Impegno / capacità / attività per la realizzazione;
- Misurazione e analisi del processo;
- Verifica dell'implementazione del processo.

Per ciascuna pratica, quale articolata in sotto-pratiche, è presente una descrizione dettagliata dal CMMI e, per raggiungere un KPA, è necessario soddisfare tutte le pratiche correlate.

Qui di seguito, le percentuali di soddisfazione dei livelli delle organizzazioni fornite dall'Università Carnegie Mellon nel 2002:



Research based on a sample of 1158 organizations

3.4 MODELLI DI PROCESSO

Con questa terminologia, facciamo riferimento alla strategia adottata da un'organizzazione per gestire il processo di sviluppo del software (come metodologie, tecniche e strumenti) che viene chiamato come "Process model" o "Software Engineering Paradigm".

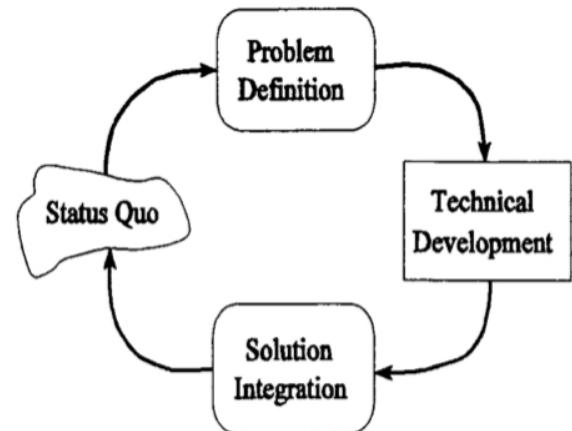
I modelli di processo sono modelli prescrittivi che definiscono un insieme distinto di attività, azioni, compiti necessari per progettare un software di qualità.

CICLO LINEARE DELLA RISOLUZIONE DEI PROBLEMI

Tutti i modelli di processo prescrittivi condividono caratteristiche comuni.

Lo stato di equilibrio di un processo di sviluppo si evolve attraverso interazioni che richiedono:

- Definizione del problema;
- Sviluppo tecnico della risoluzione;
- Integrazione della soluzione.



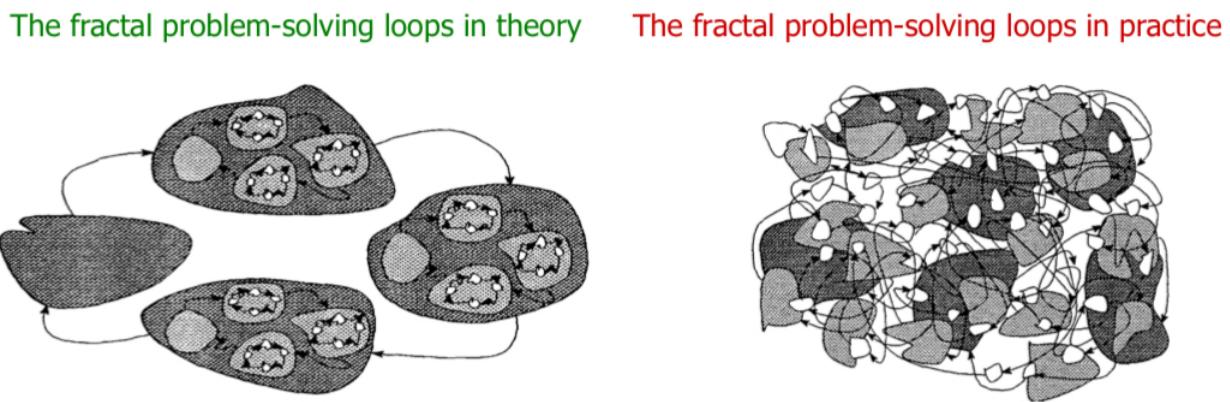
! Ogni interazione produce un nuovo stato di equilibrio che può essere modificato da successive interazioni.

CICLI DI RISOLUZIONE DEI PROBLEMI FRATTALI

Il ciclo lineare di risoluzione dei problemi si applica a molti livelli di un progetto, indipendentemente dalla tecnologia adottata o dal modello del processo.

Si applica a interi progetti, anche ai suoi sottogruppi ed ai singoli sviluppatori.

Una composizione gerarchica e ricorsiva di cicli lineari di risoluzione dei problemi, dovrebbe generare cicli frattali ideali di risoluzione dei problemi, ma non è così nella realtà, infatti:



MODELLI DI PROCESSO PRESCRITTIVI E AGILI

Alcuni dei **modelli prescrittivi** sono:

- Modello Waterfall;
- Modello a V
- Dente di sega / Dente di squalo
- Modello Incrementale;
- Prototyping;
- Modello a spirale;
- Modello Cleanroom;
- Modello dei “Unified Process”.

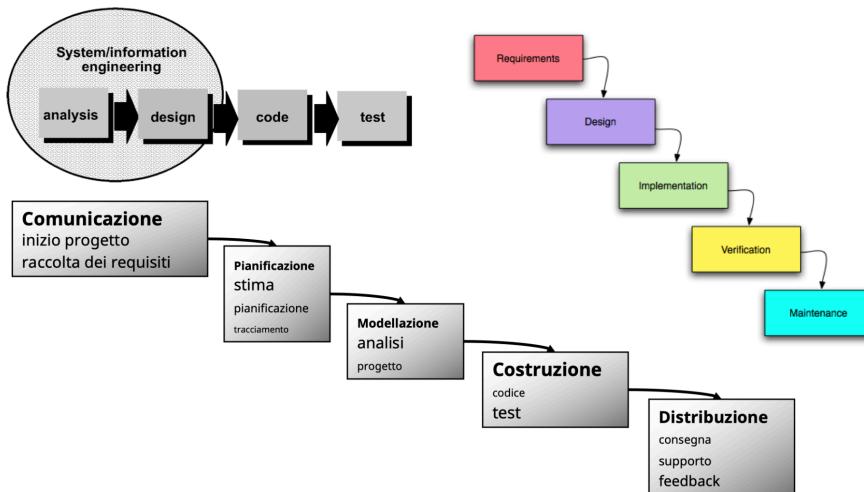
Mentre, invece, alcuni modelli di **processo agili** sono:

- Modello Extreme Programmino;
- Modello Scrum.

Andiamo ad analizzare meglio codesti modelli.

Modello Waterfall

In italiano “A cascata”, questo paradigma di sviluppo è il più vecchio e suggerisce un approccio sistematico e sequenziale, creando “una cascata”.

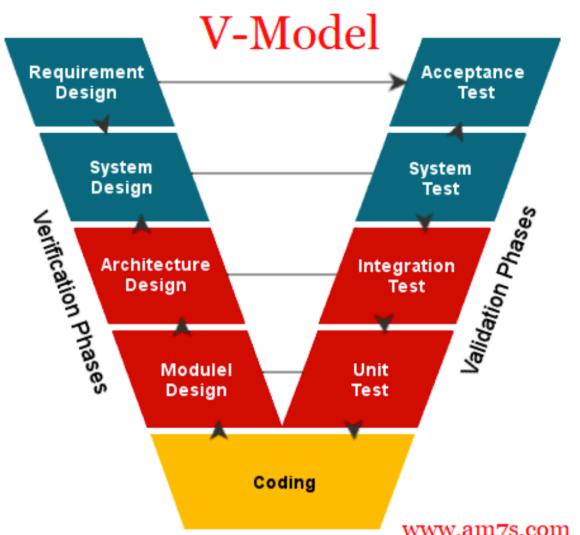


| Vantaggi | Svantaggi |
|--|--|
| Di facile comprensione. | Nella realtà, i progetti raramente seguono un flusso sequenziale. |
| Fornisce “traguardi” facilmente contrassegnabili nel processo di sviluppo. | Difficoltà per il cliente di definire i requisiti in modo esplicito. |
| Fornisce un approccio ben strutturato. | Il cliente deve avere pazienza, dato che non si riuscirà subito avere una versione funzionante del progetto. |

Modello a V

Deriva dal modello “Waterfall”, con la differenza che in ogni fase di sviluppo ha una fase di testing che viene pianificata in parallelo. È conosciuto anche come modello di verifica e validazione, distinguendo così l’attività di sviluppo dalla verifica.

| Vantaggi | Svantaggi |
|--|--|
| Di facile comprensione. | Nella realtà, i progetti raramente seguono un flusso sequenziale. |
| Milestone di facile definizione. | Difficoltà per il cliente di definire i requisiti in modo esplicito. |
| Fornisce un approccio ben strutturato. | Il cliente deve avere pazienza, dato che non si riuscirà subito avere una versione funzionante del progetto. |
| I processi diventano simultanei | |
| I difetti si individuano subito e non nella fase di Testing. | |



Modello Incrementale

Utilizza il modello del “Waterfall” ma in modo iterativo.

Infatti, comprende la consegna di una serie di release chiamate “**incrementi**” che ognuna fornirà progressivamente una funzionalità.

Il primo incremento funge da “core”, successivamente avverrà la valutazione da parte del cliente e si pianificherà lo sviluppo del prossimo incremento, permettendoci così di avvicinarsi alle sue necessità.

Questo processo avviene per ogni incremento, fin quando il prodotto non sarà ultimato.

| Vantaggi | Svantaggi |
|---|---|
| Permette release intermedie. | Assume che tutti i requisiti possano essere definiti nella fase iniziale. |
| Il piano del progetto risulta essere più complicato, ma alcune attività possono essere svolte in contemporanea. | Assume che i requisiti non cambino nel tempo. |

Paradigma Prototyping

Codesto è un processo di creazione di una versione incompleta di quello che sarà il programma, con lo scopo di costruire un prototipo da migliorare.

! Questo processo è conveniente nello sviluppo di sistemi in cui vi è molta interazione con l'utente.

Le attività principali di questo processo sono:

- La comunicazione con cliente;
- Creazione di un “Quick Plan”;
- Modellazione del programma e del suo quick design;
- Costruzione di un prototipo del programma;
- Distribuzione e feedback.

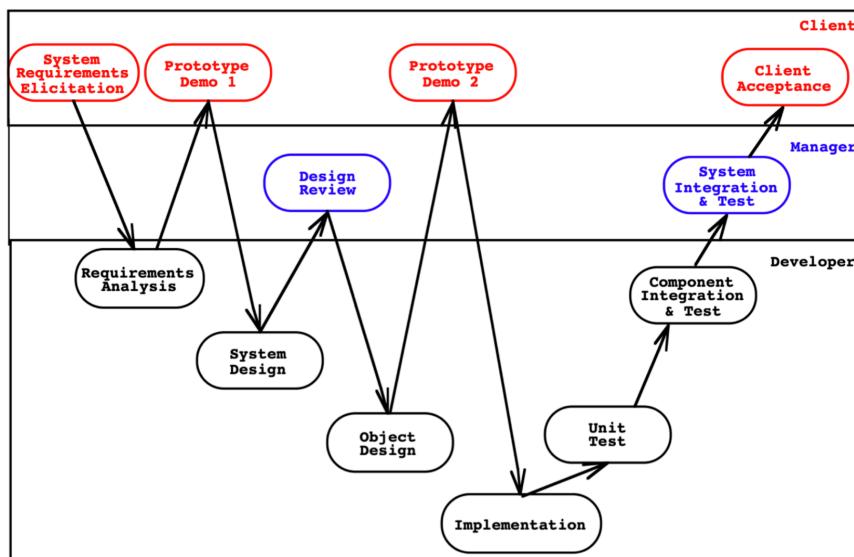
| Vantaggi | Svantaggi |
|--|--|
| Riduce i tempi e costi, perché si capisce presto cosa desidera il cliente e si evitano cambiamenti futuri. | L'analisi potrebbe essere insufficiente, perché gli sviluppatori si concentreranno maggiormente sui prototipi e meno sull'analisi completa del progetto. |
| Maggior partecipazione del cliente, così da permettere un feedback più completo. | L'utente può avere confusione tra i vari prototipi creati e la versione finale. |
| | Si impiega molto tempo nella creazione dei prototipi |

Modello a dente si sega / squalo

In codesto modello si unisce il “Waterfall” e il “Prototyping”, infatti comprende la creazione di più prototipi dimostrativi, con cui si “torna” sempre al livello del cliente, creando così graficamente dei denti.

Modello a denti di squalo

User's Understanding
Manager's Understanding
Developer's Understanding



Modello a spirale

Codesto modello può comprendere altri modelli, come casi particolari.

Ogni ciclo della spirale include le seguenti attività, denominate come “regioni”:

- Comunicazione col cliente;
- Pianificazione;
- Analisi dei rischi;
- Ingegneria;
- Costruzione e release;
- Valutazione dell’elaborato col cliente.

Attualmente questo modello è usato in grandi progetti (come da microsoft) e necessità di costante revisione. È il modello più usato tra quelli iterativi.

| Vantaggi | Svantaggi |
|--|---|
| Maggior gestione sintattica del rischio, dato che gli aggiustamenti progressivi possono essere fatti facilmente, grazie all’iterazione | È più complesso, rispetto ad altri modelli iterativi, ed è più difficile conoscere in anticipo quante iterazioni si necessiteranno. |
| Le stime diventano sempre più realistiche, perchè si riescono prima ad individuare i problemi. | È difficile garantire la sincronizzazione tra i clienti e gli sviluppatori per ogni giro della spirale. |
| Gli ingegneri del software possono iniziare subito a lavorare sul progetto. | |
| È più facile gestire i cambiamenti durante lo sviluppo. | |

4. MODELLI DI SVILUPPO AGILE

4.1 MANIFESTO DELLO SVILUPPO AGILE

“Stiamo scoprendo modi migliori di sviluppo software,
lavorandoci e aiutando gli altri a farlo”

Attraverso questo percorso siamo arrivati a valutare:

- **Individui e interazioni su processi e strumenti;**
- **Software funzionanti con documentazione completa;**
- **Collaborazione con i clienti nella negoziazione del contratto;**
- **Rispondere al cambiamento seguendo un piano.**

4.2 I METODI AGILE

Questi metodi sono stati concepiti per superare le debolezze della convenzionale
ingegneria del software.

Possono portare a grandi vantaggi ma:

- **Non sono applicabili a tutti i progetti / prodotti / situazioni;**
- **Non è opposto alla pratica dell'ingegneria del software.**

 Possono essere applicati come una filosofia superiore per un lavoro.

EXTREME PROGRAMMING (XP)

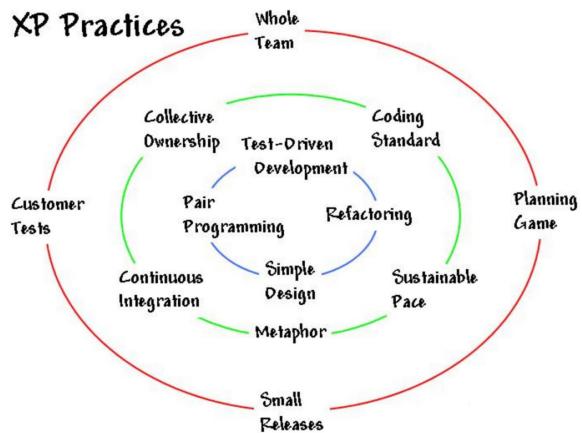
È il modello più diffuso, si basa su:

- Semplicità / coraggio / comunicazione e feedback continuo tra ingegneri, manager e clienti.
- Insieme di pratiche:

Le quali sono ben integrate in questo modello, quindi una singola pratica non può essere applicata autonomamente ad altri modelli di sviluppo:

Pratiche di programmazione estreme:

| | | | |
|---------------------------------|---|--|--|
| Whole Team | Ci si basa sulla produzione di una serie di piccole release che devono passare tutti i test definiti dall'utente. | Release Game | Iteration game |
| Planning Game | I programmatore <u>lavorano a coppie / gruppi con design "semplici"</u> e un <u>codice testato con precisione</u> , migliorandone costantemente il design per mantenerlo adatto ai bisogni. Si usano due tipi di planning: | Il cliente presenta le caratteristiche desiderate. I programmatori stipulano la sua complessità e il cliente stilera il piano per il progetto. | Ogni 2 settimane, vengono fornite nuove istruzioni al team e, alla fine di ogni iterazione, si deve consegnare una nuova versione. |
| Piccole Release | Il Team rilascia software testati e funzionanti ad ogni iterazione, quindi sarà visibile dal cliente e dagli utenti finali. | | |
| Customer Test | Assieme ad ogni caratteristica richiesta, <u>il cliente fornisce anche un test automatizzato per la approvazione del software</u> . | | |
| Design "Semplice" | <u>Si inizia il software con un "design semplice"</u> e lo si cerca di mantenere tale durante le varie versioni, adattandolo alle attuali funzionalità. | | |
| Refactoring | <u>Processo in cui si focalizza la rimozione di duplicati e sulla coesione del codice</u> , assicurandosi (attraverso testing) che non ci siano problemi, se si evolve il design. | | |
| Pair Programming | I programmatore <u>lavorano a coppie sulla stessa macchina</u> , questo per poter avere due menti che agiscono sui stessi elementi per avere <u>"doppia verifica"</u> . Questi vengono scambiati tra di loro per garantire la comunicazione. | | |
| Sviluppo basato sul Testing | <u>Questo modello è basato sul feedback</u> , quindi viene <u>tutto testato immediatamente</u> . | | |
| Integrazione continua | I team mantengono il sistema completamente integrato in ogni momento, infatti viene compilato 8/10 volte al giorno, diversamente dalla compilazione tradizionale (settimanale). Nei processi tradizionali, l'integrazione continua è meno frequente, così causando un minor "affiancamento" al sistema. | | |
| Proprietà collettiva sul codice | <u>Lavorando in coppia, le modifiche sul codice sono considerate come eseguite da entrambi</u> , a differenza della programmazione individuale. | | |
| Standard di Codice | I team seguono uno standard per la scrittura del codice, permettendo così che il codice sembri scritto di una persona e, quindi, risulti essere familiare a tutti loro. | | |
| Metafora | I team sviluppando una visione comune di come il programma lavora, chiamata "Metafora". | | |
| Ritmo Sostenibile | <u>Si usa il 40-hours-week</u> , ovvero un ritmo che può essere sostenuto indefinitamente, permettendo così ai team di massimizzare la produttività. | | |
| User Stories | Al posto della documentazione, vengono usate "User stories" scritte dall'utente in 1/2 frasi, esprimendo cosa si aspetta dal sistema, assegnandoli una priorità. Ultimi per stimare tempi e costi di produzione. | | |
| 13Esimo Principio | Riunioni in piedi di 15 minuti, dove ci si mette in cerchio e si esprime: cosa è stato fatto ieri, cosa si farà oggi e quali problemi si sono riscontrati, per poi formare le coppie. | | |



| Vantaggi | Svantaggi |
|---|--|
| Buona modalità per approcciarsi ai cambiamenti nei requisiti. | Il cliente richiede i cambiamenti in modo informale, mentre nei processi più formali vi è un comitato di controllo dei cambiamenti dei processi. |
| Fornisce buoni risultati in termini di qualità, grazie al testing giornaliero e una integrazione costante. | Per essere efficace, richiede strumenti di sviluppo complessi per la programmazione, il testing e la ristrutturazione del codice. |
| Porta ad un senso collettivo di responsabilità e favorisce la coesione tra i team. | Alcuni definiscono la “proprietà collettiva” una forma pericolosa di socialismo. |

MODELLO AGILE VS MODELLO WATERFALL

| Agile | Waterfall |
|--|---|
| <u>La fase di testing è completata nella stessa interazione della programmazione.</u> | Avviene sempre una fase separata di testing dopo una fase di build. |
| Modello Agile | |
| Vantaggi | Svantaggi |
| <u>Al cliente viene mostrato il progetto dall'inizio</u> , per poter permettere scelte durante lo sviluppo. | <u>Funziona meglio quando il team si dedica solo ad un progetto.</u> |
| <u>Il cliente</u> , lavorando col team durante la creazione, sente un maggior senso di proprietà. | <u>Non tutti i clienti potrebbero aver interesse ad essere coinvolti costantemente.</u> |
| <u>Se è più importante rilasciare una funzione in breve tempo</u> , questo modello permette di produrre velocemente una versione base da ampliare poi. | <u>Non sempre si riesce ad andare tempo con le interazioni</u> , anche data le probabili richieste future del cliente. |
| <u>Lo sviluppo è più incentrato sull'utente</u> , come risultato delle frequenti direttive del cliente. | <u>Questa natura iterativa può portare a frequenti refactoring</u> , se non si considera dall'inizio l'intero sistema. Senza questo elemento il sistema perderebbe qualità. |

SCRUM

È uno dei modelli agili, che è integrativo e incrementale, sviluppato da Jeff Sutherland e Ken Schwaber negli anni '90.

La metodologia è la seguente:

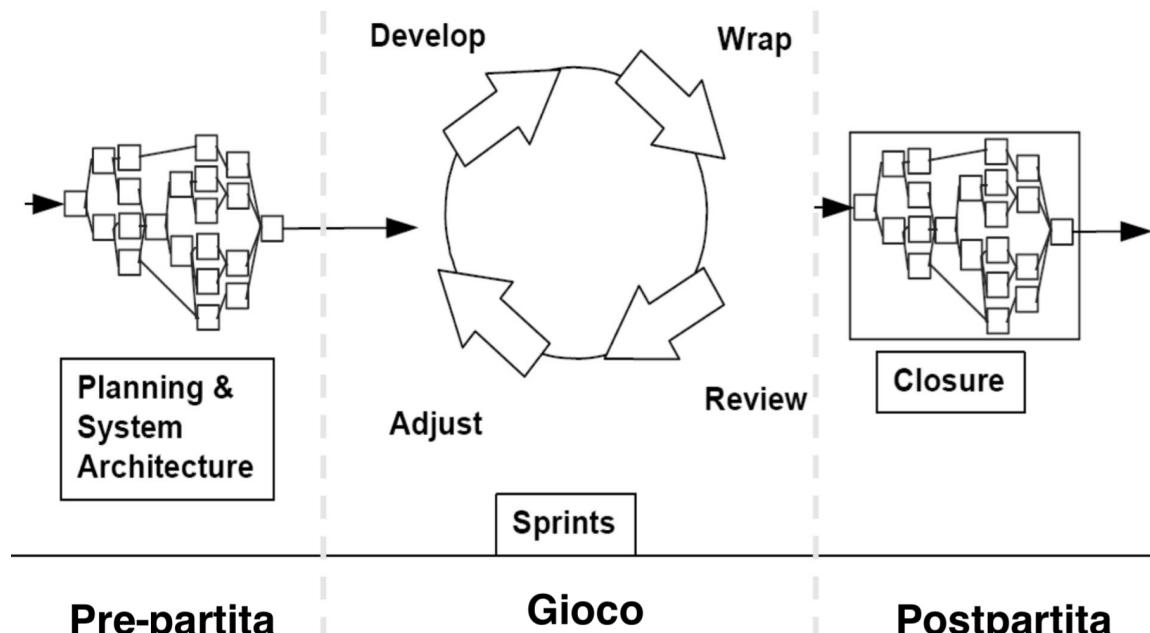
- **Il contesto è dato dall'ambiente e dai controlli;**
- **Il ciclo primario si basa sull'avanzamento;**
- Come nel rugby, che si sono evolute le regole, **si adatta all'ambiente.**
- **Lo sviluppo termina quando lo richiede l'ambiente**, considerando le necessità aziendali, la concorrenza, le funzionalità del software e il calendario.

I principi dello Scrum

Questi sono coerenti con il manifesto agile:

- **Formazione di piccoli gruppi di lavoro** (ovvero, massimo 6) che devono essere organizzati per massimizzare la comunicazione e ridurre al minimo le spese aziendali.
- **Il processo deve essere adattabile ai cambiamenti di natura tecnica che aziendale**, per garantire la produzione del miglior prodotto possibile.
- **Il processo produrrà frequenti incrementi software**, i quali possono essere ispezionati / adattati / testati / documentati e sviluppati.
- **Durante la realizzazione del prodotto**, bisogna eseguire frequenti test e aggiornare la documentazione.
- Questo modello, **permette di definire un prodotto "finito" quando lo si reputa necessario.**

Le fasi del modello SCRUM:



Come mostrato su, abbiamo 3 fasi composte da molteplici attività:

PRE-PARTITA

1. Pianificazione:

Si definisce una nuova release, basandosi su un'analisi dell'attuale backlog (ovvero una lista con le priorità dei requisiti e caratteristiche che portano valore al cliente) e sulle stime dei tempi e dei costi.

 *Se si sta sviluppando un nuovo sistema, in questa fase viene inclusa anche alla concettualizzazione, oltre che all'analisi.*

2. Architettura del sistema:

Si progetta come gli oggetti del backlog saranno implementati.

Questa fase include modifiche sull'architettura di sistema e sul design di alto livello.

PARTITA

3. Sprint:

Risultato da un insieme di attività di sviluppo condotte in un periodo predefinito, che solito va dalle 1—> 4 settimane.

Ogni interazione consiste in 4 fasi:

| | |
|----------------------|--|
| Sviluppo | Fase in cui vengono svolte le solite attività: analisi, progettazione, sviluppo, testing e documentazione. Senza obblighi di utilizzo di una metodologia formale. |
| Assemblamento | Si uniscono i componenti per avere una versione eseguibile intermedia che sia consegnabile. |
| Revisione | I team si riuniscono per presentare, revisionare i progressi conseguiti, risolvere problemi ed aggiungere oggetti al backlog. Vengono valutati i rischi e si definiscono risposte appropriate. |
| Aggiustamenti | Si consolidano le informazioni raccolte nella riunione, attraverso l'adeguamento delle componenti. |

POST-PARTITA

4. Chiusura:

Avviene quando il team di management sente che le variabili di tempo, competizione, requisiti, costi e qualità possono portare al rilascio di una nuova release da considerare "chiusa", portandola così alla produzione del suo rilascio.

! Le fasi di pianificazione e chiusura consistono in processi definiti, dove input e output sono ben chiariti ed il flusso risulta lineare.

Invece, **la fase di Sprint è considerato un processo empirico**, dove molte delle sue fasi sono indefinite e considerate come una blackbox, che richiede un controllo esterno. Per questo, per evitare caos e mantenere flessibilità, in ogni interazione si tiene conto dei controlli e della gestione dei rischi.

| Vantaggi | Svantaggi |
|---|---|
| <u>Semplice da applicare, anche in contesti caotici.</u> | <u>Definisce solo pratiche di project management</u> , ma non delle metodologie per altre pratiche come: Analisi dei requisiti / Progettazione / Testing. |
| <u>È orientato alla gestione dei cambiamenti ed a risultati concreti.</u> | |
| <u>Introduce un senso di responsabilità collettivo e favorisce la coesione in team.</u> | <u>Per risultare efficace, deve essere integrato insieme ad altri approcci</u> (come l'XP o l'Unified Process). |
| <u>Non richiede grandi trasformazioni sulla organizzazione interna delle compagnie.</u> | |

5. MODELLAZIONE E RUP

5.1 MODELLAZIONE

| È una rappresentazione approssimativa di oggetti reali che si vuole costruire.

I sistemi software d'oggi:

- Sono molto complessi,
- Possono essere collegati ad altri sistemi,
- Necessitano di continua manutenzione,
- Si evolvono rapidamente,

Per questo, gli sviluppatori necessitano di comprendere chiaramente ciò che stanno costruendo, per questo la modellazione viene d'aiuto.

? Perché alcuni sviluppatori scelgono di non modellare?

Ci sono due tipologie di risposte:

| Culturale | Temporale |
|---|--|
| I programmatore tradizionali <u>useranno tecniche / strumenti tradizionali.</u> | Gli sviluppatori credono che tale pratica li rallenti o sia inutile. |

USE CASE

| Esempio di modellazione, rappresentano una tecnica per catturare i requisiti funzionali di un sistema.

Ha le seguenti caratteristiche:

- Ogni Use Case fornisce uno o più scenari in cui vengono mostrati come interagiscono gli utenti / altri sistemi, detti **attori**, per svolgere una determinata funzione.
- Questi case, spesso, sono definiti da un analista del business e dall'utente finale.
- In codesti, si preferisce il linguaggio finale, **evitando quello tecnico**.
- Il **sistema viene trattato come scatole nere**, ovvero le interazioni sono viste esternamente, senza conoscere come siano implementate internamente.

MODELLAZIONE VISUALE

| Detta anche “Visual Modelling”, è la rappresentazione grafica del sistema, usando linguaggi grafici, che possono essere:

- **General-Purpose Modelling (GPM)**: basati su linguaggi generici, come UML.
- **Domain-Specific Modelling (DSM)**: basati su linguaggi specifici, che presentano un livello di astrazione superiore, come MatLab, SysML.

5.2 UNIFIED PROCESS

È un modello prescrittivo di processo di sviluppo software, particolarmente incentrato sulla modellazione. Oltre ad essere un processo, è anche un framework estendibile che può essere realizzato e modificato per specifici progetti.

Le caratteristiche sono:

- **Guidato dagli Use Case**, che servono per:
 - Catturare e negoziare i requisiti software;
 - Progettare l'architettura di sistema e componenti;
 - Definire i test di accettazione;
 - Pianificare il progetto.
- **Architecture-Centric**: L'architettura di un sistema costituisce le fondamenta tecniche di un progetto. Dato che nessun modello è sufficiente a coprire tutti gli aspetti di un sistema, il Unified process supporta più modelli e visualizzazioni architettonici.
- **Iterativo ed incrementale**: Le fasi del progetto sono divise in una serie di iterazioni temporali, le quali includeranno lavori comuni a diverse discipline (come analisi, requisiti, progettazione, implementazione e testing). Ogni iterazione risulta in un incremento, ovvero una release che rispetto alla precedente contiene funzionalità aggiuntive o migliorate.

RATIONAL UNIFIED PROCESS (RUP)

Versione più definita degli Unified Process, realizzata da IBM.

Si basa su **6 pratiche**, dove il **focus** è il **riuso della architettura e la riduzione dei rischi**, infatti abbiamo:

- Sviluppare il software iterativamente;
- Gestire i requisiti;
- Usare architetture a componenti, così da avere un software modulabile;
- Modellare il software in modo visivo;
- Verificare la qualità del software;
- Controllare le modifiche al software,

Questo tramite le:

4 Fasi del RUP

| 1. Inception | 2. Elaborazione | 3. Costruzione | 4. Transizione |
|---|---|--|---|
| Vengono stabiliti i limiti del progetto | Viene analizzato il dominio del problema, si stabilisce un modello architettonico, si sviluppa il project plan e si eliminano gli elementi ad alto rischio. | Vengono sviluppati i componenti e vengono testati. | Viene chiuso il progetto per essere poi consegnato. |

Andiamo a vedere nello specifico i loro criteri di valutazione:

1. INCEPTION

| **Fase detta anche “di avvio”, dove i criteri sono:**

- Concorrenza nella definizione delle stime dei costi e sviluppo temporale.
- Comprensione dei requisiti, tramite la definizione dei primi Use Cases.
- Credibilità del costo / dello sviluppo temporale / delle priorità / dei rischi e del processo di sviluppo.
- Profondità e ampiezza di qualsiasi prototipo architettonico sviluppato.
- Spese attuali contro le spese pianificate.

Come risultato di questa fase, avremmo:

- Documento di visione,
 - Piano di progetto,
 - Stima dei rischi,
 - Stima dei costi.
-

2. ELABORAZIONE

I criteri rispondono alle seguenti domande:

- La visione del prodotto è stabile?
- L'architettura è stabile?
- La demo dell'eseguibile mostra che i maggiori elementi di rischio siano stati risolti?
- Il piano per la fase di costruzione è sufficientemente dettagliato e accurato?
Si basa su stime credibili?

Come risultato avremo:

- Software Requirement Specification (SRS),
 - Software Design Document (SDD),
 - Prototipi.
-

3. COSTRUZIONE

I criteri rispondono alle seguenti domande:

- Il software è stabile e maturo abbastanza da essere rilasciato all'utenza?
- Tutti gli **stakeholders** sono pronti al rilascio? ([Sono tutti coloro che possono essere influenzati dal progetto, o che a loro volta possono influenzare il progetto stesso.](#))
- Le spese fatte finora, rispetto a quelle pianificate, consentono un margine di guadagno?

Come risultato avremo: Release beta del software.

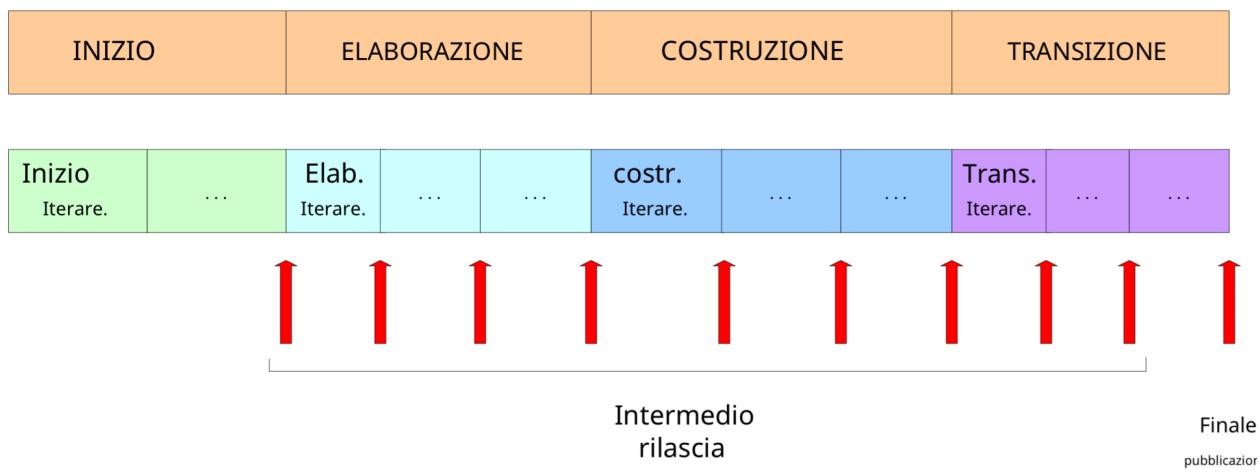
4. TRANSIZIONE

I criteri rispondono alle seguenti domande:

- L'utente è soddisfatto?
- Siamo ancora in margine di guadagno?

Come risultato avremmo: Release finale del software.

A questo punto, il progetto si considera finito e, se ci saranno nuovi cicli di sviluppo, verranno trattati come un nuovo ciclo.



Ogni fase del RUP può essere scomposta in più iterazioni, ovvero in cicli di sviluppo completo che risulta in una release (che sia interna / esterna) che può essere un eseguibile o un sottoprodotto del prodotto finale.

| Vantaggi | Svantaggi |
|--|--|
| È accompagnato da molta documentazione, come esempi e Tools. | Ha un grande impatto culturale, tecnologico e organizzativo, perché si concentra molto sulla modellazione. |
| Fornisce un'accurata specifica dei ruoli, delle attività, degli input / output delle attività e dei tools. | |

6. ANALISI DEI REQUISITI

Comprende quelle attività che vanno a determinare i requisiti di un sistema, tenendo conto dei possibili conflitti e delle richieste due vari stakeholders (tutti gli individui o i gruppi che hanno un interesse, diretto o indiretto, nel successo o nel fallimento di un progetto software)

! Parte più difficile ed è fondamentale per la buona riuscita del progetto.

6.1 ATTIVITÀ DELL'INGEGNERIA DEI REQUISITI

I processi utilizzati in codesta ingegneria variano a seconda del dominio dell'applicazione, delle persone coinvolte e dall'organizzazione che sviluppa i requisiti.

Ci sono, però, 4 **attività comuni a tutti i processi**:

1. **Sollecitazione dei requisiti,**
2. **Specificazione dei requisiti,**
3. **Validazione dei requisiti,**
4. **Gestione dei requisiti.**

1. SOLLECITAZIONE DEI REQUISITI

L'obiettivo è quello di agevolare il cliente ad esprimere quali requisiti desidera e, dato che non è una facile operazione, possono insorgere i seguenti problemi:

- **Problemi di scope:** avviene quando i limiti del progetto sono mal definiti o quando il cliente specifica dettagli tecnici non necessari, i quali possono creare confusione anziché chiarire.
- **Problemi di comprensione:** avviene quando il cliente non è sicuro di cosa desidera precisamente, oppure avendo scarse conoscenze in ambito informatico, non ha piena comprensione del dominio del problema e omette informazioni importanti.
- **Problemi di volatilità:** quando i requisiti cambiano nel tempo.

Per risolvere tali problemi, ci possiamo servire di:

Raccolta collaborativa dei requisiti:

Esistono tecniche più efficienti della classica “intervista”, infatti esistono diversi approcci che puntano ad una “raccolta collaborativa” (come FAST, JAD, ecc.) ed ognuno si avvale di uno scenario leggermente diverso, ma tutti seguono le stesse linee guida:

- **Le riunioni sono condotte e frequentate** da ingegneri del software, clienti ed altri stakeholder.
- **Si decide un'agenda che riesca a coprire tutti i punti più importanti**, ma che riesca anche ad incoraggiare un libero flusso di idee.
- **L'incontro è controllato da un “facilitatore”**, ovvero un cliente / sviluppatore / estraneo che agisce da arbitro.
- **Si definisce un “meccanismo di definizione”** (come fogli di lavoro / chat room / forum)

! L'obiettivo è quello di definire il problema, proporre soluzioni, negoziare approcci diversi e specificare un insieme iniziale di requisiti in un ambiente piacevole.

QFD – Quality Function Deployment:

È una tecnica che traduce le esigenze del cliente in requisiti tecnici.

Tale tecnica identifica tre tipi di requisiti:

| Requisiti Normali | Requisiti Previsti | Requisiti “Exciting” |
|--|--|---|
| Rispecchiano gli obiettivi dichiarati durante gli incontri con il cliente, se sono presenti, allora questo ultimo è soddisfatto. | Sono attesi ma non dichiarati esplicitamente, come che il software sia performante. La loro assenza può essere fonte di insoddisfazione. | Che rispecchiano funzionalità che vanno oltre le aspettative del cliente, per questo è soddisfacente per questo ultimo. |

Comunicazione:

L'analisi dei requisiti è un compito difficile anche perché la comunicazione può essere problematica tra le parti interessate.

Per questo, l'università del Texas ha evidenziato delle linee guida per rendere più efficace la comunicazione tra i membri del team.

| Come i membri del team devono parlare agli analisti. | Come gli analisti devono parlare ai membri del team. |
|--|---|
| Non essere timidi | Incoraggiare i membri a parlare, |
| Per aiutarli ad capire come viene svolto il lavoro, bisogna fornirli esempi. | Porre domande su come gli utenti svolgono il loro lavoro. |
| Richiedere le funzionalità che semplificano il lavoro, senza darli per scontati. | Chiedere chiarimenti e farsi ripetere i punti dell'utente per essere sicuri di aver tutto chiaro. |
| Distinguere le funzionalità necessarie da quelle desiderate. | Evitare spiegazioni tecniche / gergo informatico. |
| Stabilire le priorità da chiarire all'analista. | Menzionare i problemi che si notano. |

2. SPECIFICA DEI REQUISITI

Avviene tramite l'SRS, ovvero un documento che contiene una descrizione completa del comportamento del sistema da sviluppare, combinando il linguaggio naturale con i modelli grafici (come avviene nei UML).

Un SRS include due tipi di requisiti:

| Requisiti funzionali: | Requisiti non funzionali: |
|--|--|
| Detti tali l'insieme di casi d'uso che descrivono le interazioni che gli utenti avranno col software, quindi sono i requisiti di alto livello. <u>Per ogni requisito di alto livello, si chiariscono input e output e ognuno può essere identificato da un insieme di funzioni.</u> | Vengono descritti altri requisiti "qualitativi" che impongono vincoli alla progettazione o implementazione (come i requisiti di prestazione, standard di qualità e vincoli di progettazione). <u>Indicano caratteristiche del sistema che non possono essere espresse come funzioni (come la manutenibilità, portabilità, usabilità...)</u> |

Secondo gli standard IEEE, la specifica dei requisiti software deve essere:

| Corretta | Completa | Non ambigua | Consistente |
|--|---|---|--|
| Deve identificare con precisione condizioni e limitazioni delle situazioni che si incontreranno. | Deve definire tutte le situazioni del mondo reale che si incontreranno, le risposte a tali situazioni ed i riferimenti a tutte le figure / tabelle / diagrammi. | Deve essere interpretabile in un solo modo. | Deve non avere requisiti che siano in conflitto. |

Il documento SRS

Funge da contratto tra il team e il cliente. Una volta che il cliente accetta tale documento, il team inizia a sviluppare il prodotto, secondo i requisiti stilati nell'STS e, si considererà accettabile quando tutti questi saranno soddisfatti.

Il documento vede il programma come una "scatola nera", ovvero i dettagli interni del programma non sono noti, ma si vedono solo come input / output.

Ci si concentra su "cosa bisogna fare" evitando il "come".

Gli obiettivi di tale documento sono i seguenti:

- Facilitare le revisioni;
- Descrivere lo scopo del lavoro;
- Fornire un riferimento agli sviluppatori;
- Fornire un framework per fornire i casi d'uso primari e secondari;
- Includere funzionalità per le esigenze dei clienti;
- Fornire una piattaforma per il perfezionamento in corso.

In questo documento si possono verificarsi **errori** come:

- **Overspecification**: quando viene spiegato troppo e con termini tecnici, descrivendo il come.
- **Contraddizioni**: quando viene spiegata la stessa cosa in diversi modi.
- **Ambiguità**: se vengono usati modi di dire o si dicono aspetti non quantificabili.
- **Riferimento in avanti**: quando si punta a requisiti che saranno definiti più avanti nel documento.

3. VALIDAZIONE DEI REQUISITI

Tale fase esamina le specifiche per garantire che:

- Tutti i requisiti siano dichiarati in modo non ambiguo;
- Non siano presenti incoerente / omissioni / errori;
- I prodotti di lavoro siano conformi agli standard stabiliti per il processo, progetto e per il prodotto.

! Il meccanismo primario per la validazione è la revisione formale tecnica, eseguita da tutti gli stakeholders.

4. GESTIONE DEI REQUISITI

Dopo la validazione (3), vi è tale fase che consiste in un insieme di attività che aiutano a identificare, controllare e tracciare i requisiti e le modifiche che potrebbero avvenire successivamente.

Le sotto fasi sono le seguenti:

1. **Identificazione dei requisiti:**
 - A. Dove ad ognuno viene assegnato un identificativo univoco;
2. **Si realizzano tavole di tracciabilità, che possono essere:**
 - A. **Delle caratteristiche:** mostra come i requisiti si riferiscono a caratteristiche importanti del prodotto.
 - B. **Delle sorgenti:** indica l'origine di ciascun requisito.
 - C. **Delle dipendenze:** indica come i requisiti siano collegati tra di loro.
 - D. **Del sottosistema:** classifica i requisiti in base al sottosistema che governano.
 - E. **Delle interfacce:** mostra come i requisiti si relazionano con le interfacce di sistema interne / esterne.

7. PIANIFICAZIONE DEL PROGETTO

7.1 PROJECT MANAGEMENT

Pratica per iniziare a pianificare, eseguire, controllare e terminare il lavoro di un team con l'obiettivo di raggiungere obiettivi seguendo criteri specifici in un tempo stabilito.

I vincoli saranno quindi:

- **Costo**: sia monetario e sia delle risorse;
- **Tempo**: calendarizzazione delle risorse con stabilimento dell'inizio e fine ipotetica.
- **Obiettivi**: definendo i requisiti dell'utente e qualitativi.

I team di sviluppo comprendono diverse tipologie di membri:

- **Analisti del business e dei requisiti**: che hanno il compito di parlare con gli utenti e pianificare il comportamento e requisiti del software.
- **Progettisti e architetti**: che hanno il compito di pianificare soluzioni tecniche al problema.
- **Programmatori**: che scrivono il codice.
- **Tester**: che verificano che il software soddisfi i requisiti e i comportamenti pensati.

RUOLO DEL PROJECT MANAGER:

Pianifica e guida lo sviluppo del software, infatti:

- è responsabile di identificare utenti e stakeholders e determinare le loro necessità.
- Coordina il team, assicurandosi che ad ogni task sia assegnato un ingegnere del software con le competenze adatte.
- Deve essere famigliare con tutti gli aspetti dell'ingegneria del software.

I suoi compiti sono i seguenti:

- Capire e comprendere lo scopo del progetto.
- Scegliere quale modello di sviluppo usare.
- Sviluppare il calendario del progetto.
- Pianificare l'utilizzo corretto delle risorse.
- Iniziare l'analisi dei rischi.
- Identificare quali documenti deve produrre.
- Far partire il progetto.

DOCUMENTO DI VISION E SCOPE

Il project manager deve identificare e parlare con gli stakeholder e mostrare che si comprendono i loro bisogni attraverso codesto documento, il quale solitamente segue il seguente indice:

1. **Dichiarazione del problema:** dove vengono contenuti:

- Background del progetto,
- Stakeholders,
- Utenti,
- Rischi,
- Assunzioni (Sono affermazioni che, seppur non garantite, si considerano vere al fine di pianificare e realizzare il software.)

2. **Visione e soluzioni:** dove sono contenuti:

- Dichiarazione della visione;
- Lista delle feature.

7.2 PROJECT PLAN

Questo “**Piano**” consiste in:

- Statement of work,
- Lista di risorse,
- Scomposizione del progetto,
- Calendario,
- Piano dei rischi.

Andiamo a vederli nel dettaglio:

STATEMENT OF WORK (SOW)

Descrizione dettagliata di tutti i prodotti che verranno creati durante la vita del progetto, che include:

- Lista di feature che verranno sviluppate;
- Descrizione di ogni prodotto intermedio;
- Stima dell'effort (fatica) che sarà necessaria per ogni prodotto.

RESOURCE LIST

Come espresso dal nome, si elencheranno tutte e qualsiasi risorsa limitata necessaria al progetto, come le persone, gli elementi hardware e le stanze usate.

! Questa lista **dove dare un nome a qualsiasi risorsa, una breve descrizione, la disponibilità e il relativo costo.**

SCOMPOSIZIONE DEL PROGETTO

In questa fase, si deve definire:

- Una **Work Breakdown Structure (WBS)**, ovvero una lista di tutte le task da svolgere, che siano organizzate in una struttura gerarchica e visualizzate graficamente.
- **Stima dell'effort per ogni attività.**
- **Schedule per ogni attività.**

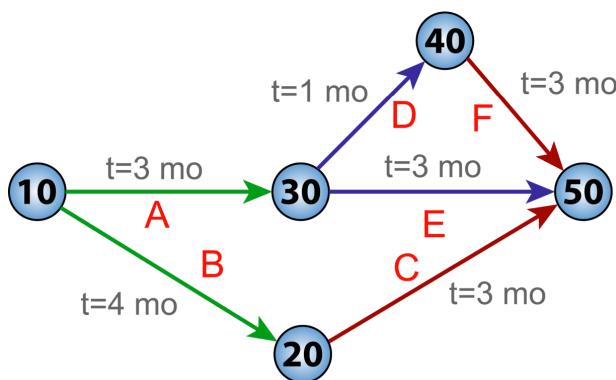
CALENDARIZZAZIONE

Per pianificare, abbiamo visto **due diagrammi**:

a. Diagramma di Pert

Usato prima che il progetto inizi, per pianificare e determinare la durata di ogni task e stimare la durata minima complessiva.

Si avvale di un grafo orientato aciclico, dove i nodi rappresentano eventi / task del progetto, mentre gli archi rappresentano le dipendenze tra di esse.



Per creare tale diagramma, serve:

- Elencare i task;
- Individuare le dipendenze tra i task;
- Disegnare nodi / frecce basandosi sui punti precedenti;
- Per ogni task, bisogna stimare un tempo.

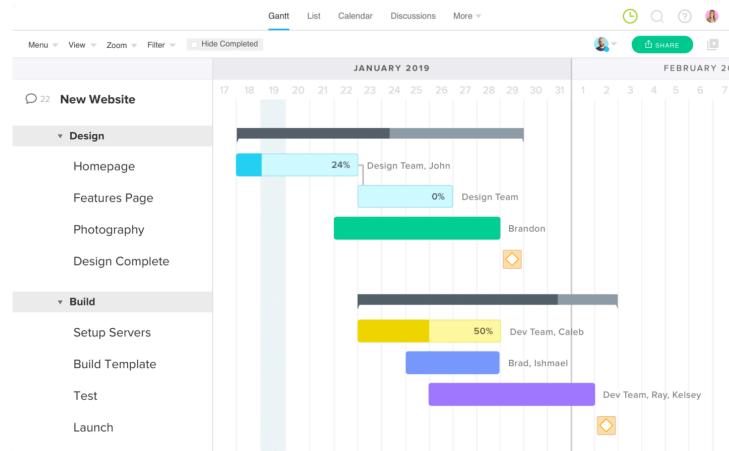
🔍 Quindi per eseguire una stima della durata, si prende il cammino più lungo dall'evento iniziale fino al finale e si sommano tutti gli archi che si incontrano.

b. Diagramma di Gantt

Usato in tutte le fasi del progetto, come strumento di monitoraggio.

Questo diagramma, creato nella prima guerra mondiale da Gantt, ad oggi viene usato per identificare le dipendenze, aumentare l'efficienza e migliorare la gestione.

Attraverso questi, è molto semplice dividere il progetto in step più gestibili e modificabili.



Per disegnarlo serve:

- Elencare task e sotto-task;
- Individuare le dipendenze tra queste;
- Creare una time-line;
- Ordinare le task;
- Assegnare le task alle persone;

Rispetto all'altro, **in questo l'andamento temporale è più scandito**.

MILESTONE VS DELIVERABLE

| | |
|--------------------|---|
| Milestone | Può essere Materiale / Intangibile Segna il raggiungimento di un punto chiave del progetto. (Quindi mostra il progresso) Permette al project manager di controllare se si rispetta il calendario. |
| Deliverable | Deve essere tangibile. Segna il completamento di una fase di progetto, permettendo così anche al cliente di verificare il suo stato. |

Spesso queste sovrappongono, ma rappresentano due strumenti distinti che permettono di visionare l'avanzamento del progetto.

La differenza sostanziale è che la Deliverable è sempre concreta.

PIANO DEI RISCHI

È una lista di tutti i rischi che minacciano il progetto, insieme ad un piano per la mitigazione di essi.

I membri che partecipano ad una sessione di pianificazione dei rischi, sono scelti dal project manager, dove si occuperanno di:

- **Brainstorming dei potenziali rischi;**
 - Ad ognuno di questi viene assegnata una probabilità e l'impatto che ci si aspetta;
- **Si costruisce un piano.**

8. STIMA TEMPORALE E DEI COSTI

STIMA TEMPORALE

Il Project manager deve avere una aspettativa sulla durata temporale del progetto, ciò avviene attraverso il confronto con gli Stakeholders e con il Team.

Per poter far tale stima, il **Project manager** deve avere:

- La **WBS** (Scomposizione strutturata del lavoro) e l'**effort** per ogni task;
- Una lista di **assunzioni** fatte per codesta stima;
- Il **consenso del team** riguardo l'accuratezza della stima.

 Per un maggiore approfondimento sulla WBS:

https://docs.google.com/document/d/1KFIR-MrN_ZKKBmys1XP0HD8y--9ZCqGP-E_qE2X_r34/edit?usp=sharing

ASSUNZIONI PER LE STIME

I membri del team fanno **assunzioni** su informazioni incomplete sul lavoro da eseguire, tali devono avere tali caratteristiche:

1. **Deve essere eseguita su una decisione ancora non presa;**
2. **I membri possono presumere le risposte**, che dovranno essere scritte, da permettere di comprendere a tutti cosa è successo.

 Tali portano al team a collaborare dall'inizio, permettendo così di parlare già di decisioni importanti che influenzano lo sviluppo.

Altre tecniche di stima:

- **Proxy Based Estimating (PROBE)**
- **COCOMO**: Dove si prevede di inserire le variabili in formule con delle euristiche.
- **Planning Game**, dove è presente l'XP, è altamente interattive e si basa sulle user stories.

 **XP**: Extreme Programming (XP) è una metodologia agile per lo sviluppo del software che mira a migliorare la qualità del codice e la reattività ai cambiamenti dei requisiti del cliente.

WIDEBAND DELPHI

Processo per scrivere il WBS e stimare l'eforo delle task.

! Tale processo si basa sul consenso univoco di tutti i membri.

È ripetibile e si compone di codesti step:

1. Scelta (o composizione) del team di stima:

- Il project manager seleziona **3 o 7 membri del team e un “moderatore”** che agisce da “arbitro” che non deve avere interesse nel progetto.

2. Primo meeting:

- Il project manager deve accertarsi che tutti i membri conoscono il procedo Delphi e abbiano letto la documentazione e siano familiari col background e i bisogni del progetto:
 - A. Il team fa **brainstorming** e scrive le assunzioni;
 - B. Si **crea il WBS** contenente 10 - 20 taskM
 - C. Il team **decide una unità di stima**.

3. Preparazione individuale:

- Ogni membro **scrive per ogni task del WBS una stima dell'effort**.

4. Sessione di stima:

- Il team raggiunge un **compromesso** sull'effort di ogni task.
- Tale sessione è divisa in **round** dove si discute per convergere su una stima:
 1. Il moderatore **raccoglie le stime e le dispone su una linea**;
 2. Attraverso la **discussione**, le risposte cominciano a convergere sempre di più a ogni round, **fino ad arrivare ad un compromesso**.

5. Assemblamento dei task:

- Il project manager lavora col team, raccogliendo le stime per ogni task e compilando la lista finale con le stime e le assunzioni.

6. Revisione dei risultati:

- Il project manager **rivede la lista finale** che si è creata.

STIMA DEI COSTI

Ciò serve per definire un prezzo per il prodotto finale.

Tale si definisce, generalmente, con:

$$\text{CostoSoftware} = \text{CostiProgetto} + \text{Profitto}$$

In tale stima, dobbiamo tenere conto di questi termini:

| Effort (Sforzo) | Durata | Costo |
|--|---|---|
| Tempo totale di lavoro. Le unità di misura sono le “giornate-uomo” / “settimane- uomo” ecc. (Una giornata-uomo equivale a una giornata lavorativa completa (solitamente 8 ore) dedicata da un singolo individuo a un’attività specifica all’interno del progetto.) | Ovvero il tempo di calendario necessario per completare una attività. La unità di misura sono ore / giorni / ecc. | Inteso come monetario per completare una attività. |

Il costo si compone nello specifico di:

| Costo dell'effort | Costo del Hardware necessario. | Costi di viaggio e Training | Altri |
|--|-----------------------------------|--------------------------------|--|
| Ovvero gli stipendi, i quali compongono il fattore dominante di tale calcolo. | | | Come il costo dell’ufficio, delle bollette ed ecc. |

Questi elementi possono essere influenzati da:

| Considerazioni | | |
|----------------|-----------|-------------|
| Economiche | Politiche | Di Business |

POLITICHE E FATTORI DEI PREZZI DEL SOFTWARE

| Fattori | Descrizione |
|--|---|
| Opportunità di mercato | Una team di sviluppo può proporre un prezzo basso perché desidera trasferirsi in un segmento diverso dal mercato. L'idea di definire un prezzo "basso" può permettere di avere maggior profitto. Tale esperienza può essere utile per la nascita di nuovi prodotti futuri. |
| Incertezza della stima dei costi. | Se una organizzazione non è sicura del prezzo da proporre, può aumentare il suo prezzo per qualche motivo e sopra al profitto stimato. |
| Termini contrattuali | Un cliente può essere disposto a far mantenere i la proprietà del codice, per poterlo fare riutilizzare in altri progetti. Per questo, il prezzo può essere inferiore. |
| Volatilità dei requisiti | Se è probabile che i requisiti cambiano, il prezzo ideale per concludere un contratto può essere inferiore. Dopo l'accordo sul contratto, i prezzi possono essere elevati per soddisfare i cambiamenti. |
| Salute finanziaria | Gli sviluppatori che sono in difficoltà finanziarie possono abbassare il loro prezzo per ottenere un contratto. Meglio fare un piccolo profitto, piuttosto che cessare l'attività. |

PRODUTTIVITÀ DEGLI SVILUPPATORI

Le unità di misura della produttività degli sviluppatori possono essere basate su:

| Dimensione | Funzionalità |
|--|----------------------------------|
| Come ad esempio le LOC (Linee di Codice) | Come ad esempio i Function Point |

LOC - Linee di Codice:

Queste possono essere **usate per stabilire i mesi-uomo** (quindi mesi composti da ore lavorative) necessari.

Esiste una sua variante, che è la DSI (Integrazione della scienza dei dati) dove vengono considerati solo le dichiarazioni e le istruzioni.

| Vantaggio | Svantaggi | | |
|-------------------|---|---|--|
| Semplice da usare | È definito sul codice e non tiene conto delle specifiche e dei documenti prodotti | Basato sulla lunghezza, quindi non considera la complessità o le funzionalità | È Language-Dependent (si riferisce a qualcosa che è specifico per un particolare linguaggio di programmazione o che funziona solo con quel linguaggio.) |
| | Un codice scadente può produrre un valore LOC non buono. | Se il codice varia spesso, non è un buon indice di produttività | |

+ è basso il livello del linguaggio, - è produttivo il programmatore.

 Tale strumento ha riportato che i programmati che scrivo un codice più dettagliato sono più produttivi di quelli che scrivono in modo compatto.

FUNCTION POINTS (FP):

Tali si utilizzano in 2 momenti:

1. **Prima di iniziare:** per stimare le dimensioni del progetto.
2. **A posteriori:** per sviluppare metriche sulla produttività per i function points come:

| Bug | Difetti | Soldi spesi | Pagine di documentazione | FP per mese-uomo. |
|-----|---------|-------------|--------------------------|-------------------|
|-----|---------|-------------|--------------------------|-------------------|

Tali Function Points **si basano su un peso, che va da 3 e 15**, sul numero dei seguenti parametri:

| Input | Output | Interazione utente | File logici interni | Interfacce esterne |
|-------|--------|--------------------|---------------------|--------------------|
|-------|--------|--------------------|---------------------|--------------------|

Moltiplicando ogni numero di ogni elemento, otteniamo gli **UFC** (Unadjusted Function Count), i quali **poi devono essere moltiplicati per i 14 Factors FI** (ovvero, fattori di complessità), che variano tra i 0 e 5 punti.

Il calcolo finale, avviene così:

$$FP = UFC * TFC$$

Dove:

$$TFC = 0,65 + 0,01 * SommaDei14FI$$

🔍 Spiegazione approfondita qui:
https://docs.google.com/document/d/1elpODY_Q50AblmJ5NHVoJwGJDOHX3xyGype9e677fkE/edit?usp=sharing

| Vantaggi | Svantaggi |
|---|--|
| Essendo non limitato dal codice, ma necessita di una specifica più dettagliata, è disponibile dall'inizio | Ignorano la qualità del prodotto finale. |
| È indipendente dal linguaggio. | Si basano su un conteggio soggettivo. |
| Essendo più generali, sono più accurate dei LOC | È difficile da rendere il calcolo del conteggio automatizzato. |

FP + LOC

I FP possono essere usati per stoccare le LOC, guardando il numero medio di LOC per FP in base al linguaggio.

$$LOC = AVC * Numero di FP$$

AVC è il numero medio di LOC per FP e, questo, dipende dal linguaggio.

8.1 STIMA DELLA DIMENSIONE DEL SOFTWARE

! Tutti i parametri basato su unità di tempo sono errati, dato che non tengono conto della qualità del prodotto che si va a realizzare, infatti la produttività può essere aumentata a discapito della qualità.

Tale avviene eseguendo la media su 3 punti:

| Dimensione ottimale S(opt) | Dimensione più probabile S(ml) | Stima pessimistica S(pess) |
|----------------------------|--------------------------------|----------------------------|
|----------------------------|--------------------------------|----------------------------|

Il calcolo quindi sarà il seguente:

$$S = [S(opt) + 4 * S(ml) + S(pess)]/6$$

METODI DI STIMA

Ne abbiamo diversi:

1. Stima algoritmica;
2. Giudizio degli esperti;
3. Stima per analogia;
4. Legge di Parkinson;
5. Pricing to win;
6. Stima top-down e bottom-up.

! La stima deve basarsi su più metodi, così da controllare di avere risultati simili.

1. Stima Algoritmica:

Utilizza una formula con variabili calibrate sulle esperienze pregresse:

$$Effort = A * Dimensione^B * M$$

Dove:

| | |
|---|--|
| A | È una costante che non dipende dall'organizzazione |
| B | Sta a indicare la non linearità dell'effort rispetto alla dimensione. |
| M | È un moltiplicatore che dipende dagli attributi del prodotto e del processo. (Solitamente si guarda la dimensione del codice e si usano i LOC). |

La dimensione effettiva si conosce solo quando il progetto è finito, dato che dipende da diversi fattori come:

| | | |
|---|------------------------------|--------------------------|
| L'uso di COTS (Commercial off the Shelf), ovvero i componenti già pronti. | Linguaggio di programmazione | Sistema di distribuzione |
|---|------------------------------|--------------------------|

Durante lo sviluppo, la stima totale e dell'effort diventano sempre più accurate.

2. Giudizio degli esperti:

Si richiede l'aiuto degli esperti nel dominio dell'applicazione e nello sviluppo software.

| Vantaggi | Svantaggi |
|---|---|
| Richiede un piccolo costo | Se non si hanno esperti nel team, non è realizzabile. |
| Può riportare una valutazione molto accurata. | |

3. Stima per analogia:

Si basa sulla storia pregressa di sviluppo di progetti simili.

| Vantaggi | Svantaggi |
|---|--|
| Se si conosce bene il progetto, risulta accurata. | Se il progetto da realizzare non è ben definito, è impossibile applicarla. |
| | Può non essere veritiera se, nel tempo, sono variati i metodi o le tecnologie di sviluppo. |

4. Legge di Parkinson:

Tale esprime che:

“Se abbiamo un totale di risorse, il progetto costerà quel totale di risorse”

| Vantaggi | Svantaggi |
|---------------------------------------|---|
| Nessuna spesa eccessiva per la stima. | Il sistema è solitamente incompleto, dato che il lavoro viene appaltato per rientrare nel budget e, quindi, riducendo funzionalità e qualità. |

5. Pricing to Win:

Si basa sull'accettare il budget proposto dal cliente.

| Vantaggi | Svantaggi |
|---|---|
| Accettando il contratto, se lo ottiene. | È difficile che il progetto venga realizzato come vuole il cliente. |

6. Stima top-down e bottom-up:

• TOP-DOWN:

Si basa su una visione black-box del sistema.

Viene usata quando abbiamo una visione parziale del sistema e, quindi, si vanno a stimare i costi:

| | | | |
|--------------|----------------|------------|-------|
| Integrazione | Documentazione | Management | Altri |
|--------------|----------------|------------|-------|

! Potrebbe sottolineare i costi singoli dei componenti.

- **BOTTOM-UP:**

Applicabile quando si conosce l'architettura del sistema a livello dei singoli componenti, dato che si vanno a sommare la stima di ognuno.

Risulta accurata se si conosce nei dettaglio tutti i moduli.

! Potrebbe sottostimare i costi dell'integrazione, documentazione e altro.

COCOMO (CONSTRUCTIVE COST MODEL)

Metodo algoritmo di stima dei costi, sviluppato nel 1981 attraverso l'analisi di 63 progetti.

Tale modello è ben documentato e indipendente (quindi non legato a nessuna software house).

Successivamente, viene ulteriormente aggiornato nel 2000, aggiungendo la considerazione di più variabili nella stima.

Distingue 3 tipologie di complessità dei progetti:

| ORGANIC: Semplice | SEMITACHED: Moderato | EMBEDDED: Complesso |
|---|---|--|
| Si ha un progetto ben chiaro in piccoli team. | Si ha una complessità media del progetto. | Progetti in cui il software è parte di un complesso, dove sono molto presenti gli hardware, software, normative e procedure operative. |

La formula è la seguente:

$$PM = Moltiplicatore * (KDSI)^E * M$$

Dove a seconda della complessità, variano il moltiplicatore e l'esponente nella formula.

Inoltre M, dipende dal prodotto di fattori di costo che COCOMO definisce in una tabella, dove riguarda anche gli attributi del prodotto, dei computer, personale e del progetto stesso.

Si basa su 4 assunzioni:

1. I progetti devono essere sviluppati col modello Waterfall;
2. I requisiti devono rimanere stabili nel corso del progetto;
3. Gli sviluppatori devono essere esperti del dominio di applicazione;
4. Diversi compiti possono essere svolti da team diversi contemporaneamente.

9. TECNICHE DI TESTING

TESTING DEL SOFTWARE

Tale processo è utilizzato per verificare la:

| Correttezza | Completezza | Sicurezza | Qualità di un software |
|-------------|-------------|-----------|------------------------|
|-------------|-------------|-----------|------------------------|

Oltre a ciò, è inclusa anche la ricerca di errori, ma si espande anche in altri aspetti.

Il testing del software, nonostante la qualità non è assoluta:

- Non **riusciremo mai a stabilire la correttezza assoluta** del software.
- **Fornisce una critica che mette a confronto lo stato e il comportamento del prodotto con una specifica.**

VERIFICA E VALIDAZIONE

Entrambe comprendono processi di controllo che assicurano che il software sia conforme alle sue specifiche e soddisfi le esigenze del cliente.

Con questi termini, ci riferiamo a due obiettivi diversi:

| Verifica | Validazione |
|--|---|
| Insieme delle attività che verificano che il software implementi correttamente le funzioni. | Insieme delle attività che si assicurano che il software sia costruito in base ai bisogni del cliente. |
| Quindi risponde alla domanda: “Stiamo costruendo bene il prodotto?” | Quindi risponde alla domanda: “Stiamo costruendo il prodotto giusto?” |

TESTING E DEBUG DEI DIFETTI E DI VALIDAZIONE

La fase di Testing è un **sotto-elemento della verifica e della valutazione.**

Ma si differenziano per il loro obiettivo:

| Testing dei difetti | Testing di Validazione |
|---|---|
| Ha l'obiettivo di scoprire gli errori e i bug. | Ha l'obiettivo di dimostrare che il sistema funziona correttamente , attraverso l'esecuzione del software come se lo usassimo normalmente. |

! Il testing ha lo scopo di scoprire gli errori, ma non localizzarli, infatti la localizzazione di questi avviene con la fase di debugging.



Più tardi nella costruzione si trova un errore, **più costerà** risolverlo.

FAILURE E FAULT

Questi sono due tipologie di errori:

| Failure | Fault (o “Bug”) |
|--|---|
| Si dice quando il software <u>non fa ciò che l'utente si aspetta.</u> Ciò è causato da un errore di Design o ad un' interpretazione sbagliata della specifica. | Quando è presente una condizione accidentale per cui il software fallisce un suo compito. Ciò è causato da un errore di implementazione. |

9.1 TECNICHE DI TESTING

Una tecnica di testing generale e completa **non esiste**, ma una di queste **si considera “buona”** quando **ha un’alta probabilità di trovare errori**.

Infatti, come esprime il Teorema di Dijkstra:

“Non esiste una procedura che ci confermi che un programma non ha Bug, ma si può solamente testare solo per input limitati”.

Esistono due tipologie di tecniche di testing:

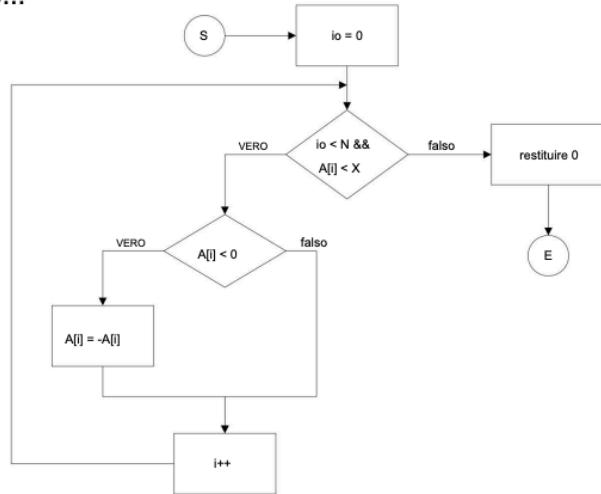
| White Box Testing | Black Box Testing |
|--|---|
| Necessità di una visione interna del codice dell’oggetto testato , infatti si basa sull’analisi del codice e si utilizza durante lo sviluppo di questo, | Necessità di una visione esterna dell’oggetto testato , basato sull’analisi del dominio degli Input e si usa quando si hanno le prime versioni software. |

WHITE BOX TESTING:

Come già detto, si ha una **visione interna del sistema e della sua struttura**, tramite l'uso di **Flowcharts**, come da esempio qui di seguito:

Viene fornita la seguente funzione...

```
int foo(int A[], int N, int X)
{
    int io = 0;
    mentre (i < N && A[i] < X)
    {
        se (A[i] < 0)
            A[i] = -A[i];
        io++;
    }
    restituire 0;
}
```



🔍 Dato che non si può testare in un programma infiniti input, si usano le tecniche di copertura.

In programmi più complessi, non si testano tutti i casi, ma si cerca di avere una ottima percentuale testata, dove la formula è la seguente:

$$Coverage = \frac{N.Test}{N.IdealeTest} * 100$$

In ogni test, si percorre un logic path (percorso logico) diverso ed esistono diverse tecniche di copertura del codice:

| | |
|----------------------------|--|
| Statement Coverage | Si selezionano un set di test che esegue ogni costrutto del programma (detto action Block) almeno una volta. |
| Branch Coverage | Si seleziona un set che attraversi ogni i flusso di controllo almeno una volta. |
| Condition Coverage | Si seleziona un set che testi ogni condizione in modo che sia vera e falsa almeno una volta. |
| Basis Path Coverage | <p>Tecnica proposta da T. McCabe nel 1976. Consiste nel creare il Flowchart del programma, per poi generare un grafo, per permetterci di ricavare tutti i path indipendenti (ovvero, quelli che percorrono almeno un nuovo arco).</p> <p>Questo testing fornisce una misura quantitativa della complessità logica del programma, che è definita dal numero massimo di path indipendenti del grafo.</p> <p>Per questo è importante avere un grafo ben strutturato, per poi poter scrivere un codice ben strutturato o, per lo meno, permetterci di aggiungere nodi unici di entrata / uscita.</p> |

BLACK BOX TESTING:

| **Non si ha accesso al codice e ci si concentra sui requisiti funzionali.**

Tale visione si basa sulla copertura del dominio di input, la quale non potrà mai essere al 100%.

Studiamo due tecniche di questa tipologia di testing:

| Partizionamento d'equivalenza | Tecnica dei Valori di confine (Boundary Value Analysis) |
|--|---|
| <p>Tecnica che divide il dominio input in classi di equivalenza, che sono definite per ogni condizione di input.</p> <p><u>Da ogni classe di equivalenza si prende un campione e, facendo così per ogni classe di equivalenza, avremmo ricoperto tutti i tipi di input.</u></p> | <p>Tecnica dove si parte con l'idea che gli errori si verificano nei "confini" del dominio, serve per identificare potenziali difetti software associati ai valori estremi o di confine degli input e degli output.</p> <p>In altre parole, la BVA si concentra sui valori minimi, massimi e appena al di fuori dei limiti validi per i dati di input e di output.</p> <p> P.S. Descrizione fornita da ChatGPT</p> |

! Anche nel caso del Black Box Testing si possono combinare tecniche per avere un testing più efficace.

GRAY BOX TESTING (Translucent Testing):

Combinazioni di metodi di Testing visti prima, infatti **combiniamo il testing interno** (con le metodologie della White Box) e **il testing esterno** (con i metodi della Black Box).

Questa unione, **porta alle metodologie della Gray Box**, che ci viene in soccorso quando abbiamo una visione limitata del sistema (come quando abbiamo visione dell'UML ma non del codice).

| Efficace per... | Non efficace per... |
|--|--------------------------|
| Casi come applicazioni web / sistemi distribuiti e verifiche di sicurezza. | Testing degli algoritmi. |

! **Utile nel integration testing** (fase in cui si verifica come i moduli software funzionano insieme come un sistema completo).

🔍 I test sono fatti dal punto di vista dell'utente finale e possono essere svolti da codesto che sia dai tester (i quali si immedesimano).

Come agisce?

| | | |
|----|---------------------------------|---|
| 1. | White Box Testing | Si usano le tecniche della White Box Testing per comprendere le basi della struttura interna dell'applicazione |
| 2. | Progettazione Test Cases | Si usa la conoscenza acquisita precedentemente per definire i casi di insiemi specifici di condizioni di input, azioni eseguite e risultati attesi utilizzati per verificare la funzionalità e l'affidabilità di un software, detti Test Cases . |
| 3. | Black Box Testing | Si usa la conoscenza acquisita fino a qui per eseguire i test "esterni" tramite i Black Box Testing. |

Le Tecniche di Gray Box Testing:

| | |
|--------------------------------------|--|
| Matrix Testing | Viene usata una matrice con tutte le variabili e valori che queste possono assumere, successivamente vengono esaminate e scritti i rischi di ogni variabile in una tabella. |
| Pattern Testing | Vengono usati i pattern di testing che sono stati usati per software simili, con la speranza di trovare errori simili. |
| Testing dell'array ortogonale | Tecnica abbastanza complicata, dove quando sono poche, vengono usate le variabili in input. Si dice "array ortogonale" la tabella dove sono presenti tutte le possibili coppie di parametri che compaiono solo una volta, quindi non hanno ripetizioni. |

FLAKY TEST

Detti anche test “**instabili o inaffidabili**”, sono **casi di test del software che producono risultati incoerenti**.

In altre parole, a volte possono passare e altre volte fallire per lo stesso input e la stessa configurazione di test.

Questa incoerenza può essere causata da una varietà di fattori, tra cui:

| | |
|---------------------------------------|--|
| Problemi di concorrenza | |
| Caching dei Dati | |
| Ambiente di testing non pulito | Finito il caching non vengono pulite le variabili d’ambiente, causando risultati falsi al prossimo test. |
| Ambiente dinamico | Tale quando, ad esempio, si testano le interfacce grafiche o si fanno chiamate asincrone. |
| Bombe a Tempo (Time Bomb) | Quando si usa il tempo della macchina o si fanno assunzioni temporali |
| Problema di Infrastruttura | Non dipendono dal tester, ma dalla connessione, versione del browser, ecc. |

Possono rendere difficile identificare e correggere i difetti reali, poiché i fallimenti dei test potrebbero non essere sempre indicativi di un problema effettivo, oppure possono non risultare, **infatti possono portare a un risultato non-deterministico**.

Come si affrontano?

1. **Bisogna segnare** quali test sono considerati “Flaky”;
2. Quando il test fallisce, **si registrano tutti i dati del test**, ovvero:
 - Log;
 - Stato corrente del sistema;
 - Screen della UI.
3. **Si ri-esamina il codice** per capire se il problema è lì presente;
4. **Si analizza** se dipende da altri fattori.

! I Flaky Test sono un problema comune nel testing del software, ma non sono inevitabili, infatti anche Google (ad esempio) né è “afflitta”.

10. STRATEGIE DI TESTING

Una strategia integra le tecniche di progettazione di test in una serie pianificata di passi che risulta efficace nella costruzione di software.

In genere, contiene le seguenti attività:

1. **Pianificazione** dei Test;
2. **Progettazione** dei Test;
3. **Esecuzione** dei Test;
4. **Raccolta e valutazione** dei risultati.

Tutte le strategie condividono queste caratteristiche:

La fase di Testing inizia sempre dal basso (quindi con le modalità “In The Small”) **per poi espandersi fino all’intero sistema** (con le modalità “In The Large”).

Vengono usate tecniche diverse in base al livello in cui ci troviamo.

In genere, **il testing è condotto dagli sviluppatori software o, nei progetti molto grandi, da un team di tester.**

SCAFFOLING - STRUTTURA DI UN AMBIENTE DI TESTING

È composto da **4 elementi**:

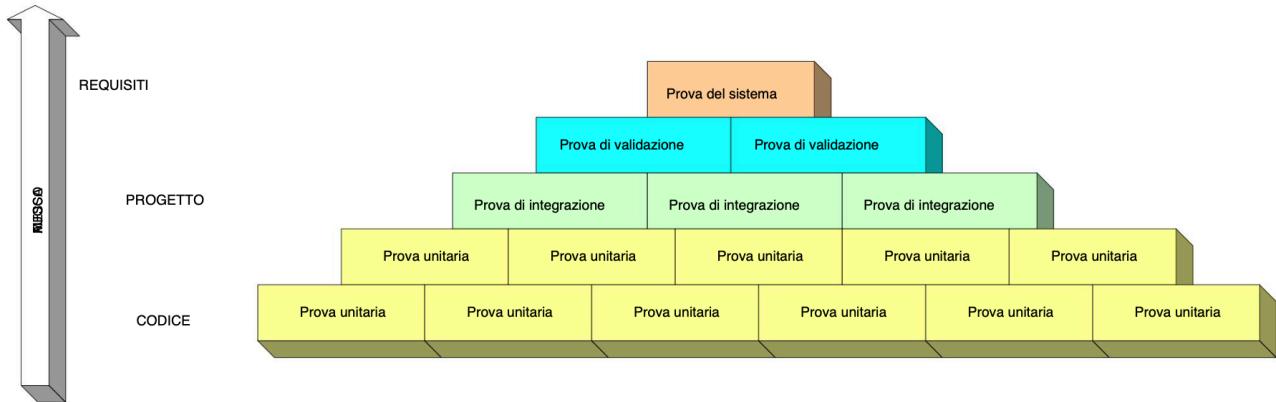
| | | |
|----|--|--|
| 1. | AUT (Application Under Testing) | Oggetto da testare. |
| 2. | STUB | Sono moduli ausiliari che rendono l'AUT esegibile. |
| 3. | Driver | Applicazione esterna dove verrà eseguito il nostro test. |
| 4. | Oracolo | Altra applicazione esterna che verifica il risultato del test. |

TEST DI REGRESSIONE

Ogni volta che si aggiunge una feature o si fa un commit, bisogna ri-eseguire tutti i test, per verificare se sono presenti nuovi bug sulle vecchie funzionalità.

A questo scopo, **si salvano gli scaffolding dei test precedenti, con i loro input e output, per poterli riutilizzare.**

LIVELLI DI TESTING DI UN SOFTWARE



Si basano su 4 fasi:

| | | |
|----|-----------------------------|--|
| 1. | Test delle Unità | Test che si concentra su ciascuna componente che non è scomponibile. |
| 2. | Test di Integrazione | Dove ci si concentra sul design e dell'architettura del software. |
| 3. | Test di Validazione | Dove si valutano se si soddisfano le richieste del cliente. |
| 4. | Test di Sistema | Dove si testano tutti i componenti insieme. |

Andiamo a visionare tali test nello specifico:

1. Test delle unità

È presente il concetto di scaffolding, infatti abbiamo:

| Driver | Stub |
|---|--|
| Nella maggioranza dei casi corrisponde al programma principale (main), il quale "accetta" di eseguire test con dei dati, trasmetterli ai componenti e stampare i risultati. | Sono programmi che simulano i comportamenti di moduli subordinati a quello testato. Può eseguire una manipolazione minima dei dati inseriti per il test, fornisce la verifica all'ingresso e restituisce il controllo al modulo sottoposto al test. |

Lo **unit testing** è ovviamente basato su tecniche di **White box testing**: ci si assicura di percorrere tutti i path indipendenti per coprire tutti gli statement.

In caso di necessità, si possono usare anche test black box.

Lo unit testing è un compito oneroso, ma si possono testare diversi moduli contemporaneamente se autonomi.

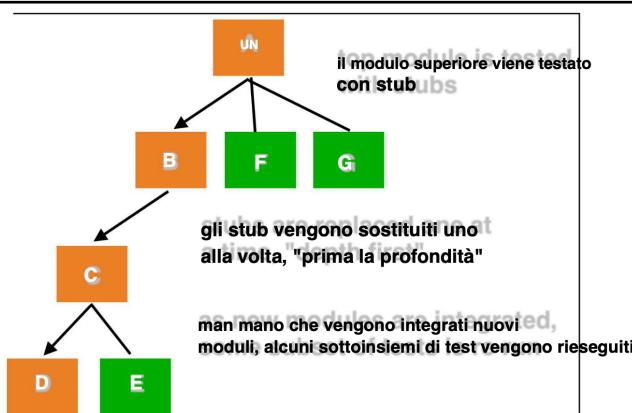
2. Test di Integrazione

È una tecnica sistematica usata per scoprire errori riguardanti l'intrecciamento tra i moduli.

L'integrazione **deve essere incrementale**, unendo i moduli man mano e, quindi, non bisogna eseguire il test nel complesso, *come avviene nell'approccio Big bang*.

Tale integrazioni può essere eseguita in diversi modi:

Integrazione Top-Down:

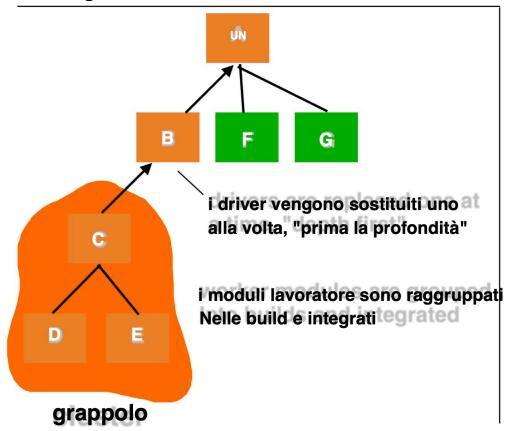


È un approccio incrementale, infatti **si integrano i moduli spostandosi dall'alto verso in basso della gerarchia, partendo dal modulo di controllo.**

Ogni volta che si sostituiscono gli Stub dei relativi moduli, bisogna risentire alcuni test.

| Vantaggi | Svantaggi |
|---|--|
| Localizzare gli errori è semplice. | Necessità di vari Stubs. |
| Possibilità di ottenere velocemente un prototipo. | I moduli ai livelli bassi sono testati in modo inadeguato. |
| I moduli "critici" sono testati con priorità. | |

Integrazione Bottom-Up



Questo approccio comprende la combinazione dei componenti di basso livello in un “cluster”, per poi essere considerati come singoli elementi successivamente.

Infatti, per questo, si andrà a sostituire un intero ramo, che sappiamo funzionare, con il cluster.

| Vantaggi | Svantaggi |
|--|---|
| La localizzazione degli errori è semplice. | I moduli “critici”, essendo testati per ultimi, possono essere soggetti a difetti., |
| Non viene perso tempo nell’attendere che tutti i moduli vengano sviluppati, a differenza dell’approccio “Big Bang” | Un prototipo iniziale non è possibile crearlo. |

Integrazione a Sandwich:

Si combinano le due tipologie di integrazioni appena spiegate, sfruttando così solo i vantaggi che hanno, permettendo anche di avere un prototipo iniziale.

2a. Smoke Integration

Viene usata quando il progetto ha un approccio “step by step” ed è molto grande, risponde a domande come: **“Il programma esegue?”, “L’interfaccia si apre?”**.

I passaggi sono i seguenti:

1. I moduli che vengono testati e che sono funzionali, vengono integrati in una “Build”, le quali contengono:

| | | | |
|-----------|----------|-------------------|---|
| Data file | Librerie | Modelli riusabili | Tutto ciò che è necessario per implementare una / più funzioni. |
|-----------|----------|-------------------|---|

2. Viene creata una batteria di test per trovare gli errori nella Build;

3. La build verrà integrata con altre build.

L’intero prodotto verrà testato ogni giorno con una Build più grande e usando altri tipi di integrazione.

| Vantaggi |
|---|
| Essendo i test condotti quotidianamente, i rischi di integrazione sono minimizzati. |
| Essendo l’approccio molto basato sulla architettura, gli errori di design sono facilmente scoperti. |

3. Test di Validazione

Permette di verificare se il programma soddisfa i requisiti richiesti e se è ragionevolmente come previsto per il cliente.

Le richieste del cliente sono definite nel **SRS (Specifiche dei Requisiti Software)**, dove è prevista una **sezione chiamata “Criteri di Valutazione”** che costituisce un approccio ai test di valutazione.

Nei test di validazione si usa il **Black box testing**, basati solitamente sugli UML.

È impossibile predire il comportamento dell'utente finale, il quale può fraintendere le istruzioni o usare strane combinazioni di dati, **per questo se si deve creare un software per un solo cliente, lo si chiama in azienda per fargli eseguire dei test di approvazione.**

Se si hanno invece più clienti possiamo avvalerci di altri due metodi:

| Alpha Test | Beta Test |
|--|--|
| Svolto in azienda da uno degli utenti finali, con uno sviluppatore che osserva e segna gli errori o problemi che sono generati con l'uso. | Il programma viene rilasciato per essere usato singolarmente dai vari utenti , che hanno l'impegno di registrare i problemi e segnalarli allo sviluppatore. |

4. Test di Sistema

Come ultima cosa si testa come il nostro sistema interagisce col mondo esterno, ovvero:

| | | |
|------------------------|----------------------|-----------------------|
| Con altre applicazioni | Con database esterni | Con hardware diversi. |
|------------------------|----------------------|-----------------------|

I test di sistema possono essere di vari tipi:

| Recovery Testing | Security Testing | Stress Testing |
|---|---|---|
| Viene forzato il software per ottenere un guasto e verificare che la recovery agisca correttamente. | Verifica che i meccanismi di protezione integrati funzionino correttamente. | Si esegue un sistema in modo tale da richiedere risorse in quantità / frequenza / volume anomale. |

11. GUI TESTING

INTERFACCIA UTENTE

Devono essere progettate per corrispondere alle competenze, all'esperienza e aspettative degli utenti per cui è progettata.

Gli utenti sono più soggetti a giudicare l'interfaccia, rispetto alle funzionalità, per questo, se questa è **mal progettata** può causare, **oltre che errori** da parte dell'utente, ma **anche il non utilizzo** dell'applicativo.

FATTORI UMANI DELLA PROGETTAZIONE DELL'INTERFACCIA

| | |
|--|---|
| Memoria a breve termine limitata | Le persone possono <u>ricordare istantaneamente 7 informazioni circa</u> , di più si rischia maggiore soggezione ad errori. |
| Le persone fanno errori | Quando le persone commettono errori, allarmi e messaggi <u>inappropriati possono aumentare lo stress</u> di questo ed aumentare la probabilità di ulteriori errori. |
| Le persone sono diverse | <u>I progettisti non dovrebbero progettare solo in base alle proprie capacità</u> , dato che gli utenti ne hanno una vasta gamma. |
| Le persone hanno preferenze di interazione diverse. | <u>Alcuni utenti preferiscono</u> più interazioni tramite immagini, mentre altre con testo. |

PRINCIPI DI PROGETTAZIONE

Si basano su:

| | |
|-------------------------|---|
| User Familiarity | L'interfaccia dovrebbe essere basata su termini e concetti orientati all'utente finale, rispetto ad uno esperto del campo. |
| Consistency | Il sistema dovrebbe mostrare un livello adeguato di coerenza grafica. |
| Minimal Surprise | Gli utenti non dovrebbero mai essere sorpresi dal comportamento del sistema. |
| Recoverability | Il sistema dovrebbe fornire una certa resilienza agli errori dell'utente e consentire all'utente di riprendersi da ciò, come tramite una funzione di annullamento o conferme di azioni "distruttive". |
| User Guidance | L'interfaccia dovrebbe fornire un feedback quando si verificano errori e fornire strutture per aiuto. |
| User Diversity | L'interfaccia dovrebbe fornire strutture di interazioni adeguate per i diversi tipi di utente. |

PERCHÉ TESTARE LA GUI?

Considerando che i moderni sistemi informatici sono progettati utilizzando l'approccio dell'architettura a più livelli, bisogna considerare quindi che le **funzionalità principali** del sistema è contenuta all'interno del livello della “**logica aziendale**” come una serie di componenti aziendali, discreti ma connessi.

Questi, **sono responsabili ad acquisire informazioni** dalle varie interfacce utente, eseguire calcoli e “transazioni” a livello di database e presentare i risultati al utente.

Tale procedura **non è semplice**, spesso **sarà necessario testare l'interfaccia** per coprire tutte le funzionalità e avere una copertura di test completa.

Per questo, ci aiuta la fase di:

TEST DELLA GUI :

| Consente di **testare la funzionalità dell'applicativo dal punto di vista dell'utente**.

Cosa Testare?

| | |
|----|--|
| A. | Controllare tutti gli elementi della GUI per: dimensioni, posizione, larghezza / lunghezza / accettazione di caratteri o numeri. |
| B. | Verificare di poter eseguire la funzionalità prevista, tramite GUI. |
| C. | Verificare che i messaggi di errore vengano visualizzati correttamente. |
| D. | Verificare che vi sia una chiara delimitazione delle diverse sezioni sullo schermo. |
| E. | Verificare che il carattere usato in un'applicazione sia leggibile. |
| F. | Verificare che l'allineamento del testo sia corretto. |
| G. | Controllare che il colore del carattere e dei messaggi di avviso siano esteticamente gradevole. |
| H. | Controllare che le immagini siano allineate e visualizzate correttamente. |
| I. | Controllare che il posizionamento degli elementi della GUI per la diversa risoluzione dello schermo. |

Tipi di Test Della GUI :

Test Manuale

| **Le schermate grafiche vengono controllate manualmente dai tester**, in conformità con i requisiti indicati nel documento dei requisiti aziendali.

Registrazione Analogica

Lo strumento di test cattura specifici clic del mouse, pressioni sulla tastiera e altre azioni dell'utente, memorizzandoli in un file per la riproduzione.

Abbiamo due tipologie di registrazione analogica:

A. Relativa:

Le posizioni degli eventi di interazione vengono registrate rispetto all'angolo superiore sinistro della finestra della applicazione.

B. Assoluta:

Vengono registrate le posizioni degli eventi, rispetto all'angolo in altro a sinistra dello schermo del sistema.

| Vantaggi | Svantaggi |
|---|---|
| Funziona con tutte le applicazioni, indipendentemente dalla tecnologia / piattaforma utilizzata. | Necessario rendere standard i fattori variabili che possono variare i risultati dei test, come risoluzione dello schermo / dimensione delle finestre ecc. |
| Non è richiesto allo sviluppatore di rendere l'applicazione testabile. | I test non sono intelligenti , quindi saranno solo una serie di gesti registrati. Si richiede una convalida umana , dato che questi test eseguono un'azione e quindi il feedback è limitato. |

Basato su Oggetti

Lo strumento di test è in grado di connettersi, a livello di codice, all'applicazione sottoposta ai test e “vedere” ciascuno dei singoli componenti dell'interfaccia utente, come un pulsante / una casella di testo e altro, come entità separate.

È in grado di eseguire operazione (quindi eseguire click o inserire testo) e leggere lo stato in modo affidabile, indipendentemente da dove si trova l'oggetto sullo schermo.

| Vantaggi | Svantaggi |
|---|---|
| Il test è più “robusto” e non risponde al fatto che gli oggetti dell'interfaccia utente si trovino in una determinata posizione. | Lo strumento di test deve avere un supporto specifico per ciascuna delle tecnologie utilizzate nell'applicazione . |
| Il test fornisce un feedback immediato quando fallisce. | <i>Per alcune tecnologie, gli sviluppatori necessitano di aggiungere strumentazione al loro codice</i> , in modo che gli strumenti di test siano in grado di “vedere” gli oggetti dell'interfaccia. |
| I test saranno meno “fragili” e richiederanno meno elaborazioni man mano che l'applicazione cambia. | Scrivere i test può richiedere più abilità , infatti è necessario essere in grado di utilizzare strumenti di spionaggio / ispezione per navigare nella gerarchia degli oggetti e interagire con i gesti elementi dell'interfaccia utente. |

Quando Usarli?

La procedura migliore consiste nell'utilizzare **la registrazione basata su oggetti**, dato che è più solida e affidabile, ma dove è possibile utilizzare **la registrazione analogica** per “colmare le lacune” dove necessario.

Sfide nel Test della GUI

| | |
|--------------------------------|---|
| 1. Repeatability | Le interfacce utente delle applicazioni cambiano spesso in modo significativo tra le versioni . Di conseguenza, durante il test di un'interfaccia utente, potrebbe essere necessario rifattorizzare ampie parti degli script di test registrati affinché funzionino correttamente con la versione aggiornata. |
| 2. Technology Support | Le applicazioni possono essere scritte utilizzando una varietà di tecnologia e possono utilizzare una varietà di libri di controllo diverse , non tutte di quest'ultime sono così facili da testare e strumenti di testa specifici potrebbero funzionare meglio per alcune rispetto ad altre. |
| 3. Stability of Objects | Le scelte degli sviluppatori possono facilitare il testing della GUI , infatti più gli oggetti hanno ID diverso ad ogni apertura della finestra e più è difficile tale test. |
| 4. Instrumentation | In alcuni casi dove lo sviluppatore deve aggiungere ai test codice speciale (chiamato strumentazione) o compilare con una libreria specifica per consentire i test , ma non ha l'accesso al codice sorgente, i test vengono limitati. |

TEST DELL'ESPERIENZA UTENTE (UX)

Gli utenti finali effettivi / rappresentati degli utenti **valutano un'applicazione per la sua facilità d'uso, appeal visivo e capacità di soddisfare le loro esigenze**.

Sempre più spesso, questi test vengono eseguiti virtualmente tramite cloud e i risultati possono essere raccolti tramite osservazioni in tempo reale dagli utenti.

I team di progetto possono effettuare **sessioni di beta testing**, dove un'applicazione completa o quasi viene resa disponibile per il test da parte degli utenti presso la propria sede, con i risultati raccolti tramite moduli di feedback.

| UXT | UAT |
|--|---|
| Riferimento ai test dell'esperienza utente | Riferimento ai test di accettazione dell'utente, ovvero è un livello di test che verifica che una determinata applicazione soddisfi i requisiti richiesti |

12. DEBUGGING

| Processo di ricerca e risoluzione di comportamenti anomali di un sistema software.

È una **fase importantissima**, infatti i programmatori ci dedicano almeno il 50% del tempo totale, dato che è una attività costosa e onerosa.

Negli IDE e framework moderni, come eclipse, il debugger è integrato,

Distinguiamo:

| Test | Debug |
|---|--|
| Viene eseguito il programma con l'intento di trovare bug. | Quando viene trovato l'errore, bisogna capirne la natura e provare a risolverlo. |

PROCESSO DI DEBUGGING

1. Inizia con l'**esecuzione di un test case**, ovvero con un insieme di istruzioni dettagliate che descrivono come testare una specifica funzione o aspetto di un software.
2. Si **valutano i risultati** e si **confrontano le performance attese** e quelle **riscontrate**.
3. Avremo *una delle due situazioni*:
 - A. **Si riscontra l'errore e lo si corregge**;
 - B. **Non si riscontra l'errore**:
 - › Se avviene ciò, il debugger potrebbe avere dei “sospetti” e quindi sarà necessario progettare un altro test case che lo aiuti ad indicare e lavorare in modo iterativo.

Approcci al debugging:

| | |
|---------------------------------|--|
| Forza bruta | Vengono inserite delle print, con la speranza di individuare dove è allocato il problema. <i>(È l'approccio più semplice, ma il più sconsigliato, dato che è più oneroso di fatica e tempo)</i> |
| Backtraking | Viene tracciato l'errore, percorrendo a ritroso il codice. Può essere usato con successo in programmi “piccoli”, dato che più righe sono presenti e più può diventare ingestibile. |
| Eliminazione delle cause | Introduce il concetto di partizionamento binario, infatti si vanno ad escludere man mano parti di codice, isolando così le potenziali cause di errore . |

DEBUGGER

Meta-programma che è capace di eseguire un altro programma con condizioni controllate.

Strumenti del debugger:

| | |
|-------------|---|
| Breakpoints | È un arresto / pausa intenzionale che è messo in atto per scopi di debug. |
| Watchpoint | Meccanismo dove l'esecuzione viene sospesa ogni volta che viene modificata una posizione di memoria specificata. Vengono chiamati anche "Punti di Interruzione" |
| Asserzioni | Espressioni booleane che vengono inserite nel codice, se queste risultano "True" continuano l'esecuzione, sennò si blocca. |

Quando il debugger incontra un breakpoint, causando l'arresto dell'esecuzione, il controllo “passa” al programmatore tramite la scelta offerta dal debugger:

- Continue:

Il programma continua l'esecuzione fino al prossimo breakpoint o finché non termina.

- Step:

Il programma continua fino alla successiva riga di codice (o ad una invocazione), distinguendo due tipologie:

- ➔ **Step into:**

Se è presente una chiamata a funzione, si entra al suo interno.

- ➔ **Step over:**

Se è presente una chiamata a funzione, si passa oltre senza debuggarla.

Record-Replay Debugging:

Tecnica per riprodurre i bug, dove l'esecuzione viene registrata e poi ripetuta, senza la possibilità di “rimandarla indietro”.

DEBUGGER REVERSIBILI

Permettono di fare “step indietro” e tengono traccia della storia dell'esecuzione.

🔍 Con codesti debugger, uno studio, ha accertato che i programmatori spendono il 26% in meno a debuggare.

Abbiamo diversi debugger reversibili, come:

- il GDB di GNU che ha introdotto i commandi reversibili dalla versione 7, rilasciata nel 2009;
- Mozilla con RR;
- Microsoft con .net.

GDB

Debugger rilasciato da GNU, dove nella versione 7, rilasciata nel 2009, è reversibile e usa la tecnica di record-replay.

Oltre ai commandi di test già citati, ovvero *step (step into)* e *next (step over)*, sono presenti altri commandi reversibili, come:

| | |
|-------------------------|---|
| Reverse-step | Esecuzione vino alla prossima riga / invocazione. |
| Reverse-next | Esecuzione fino alla prossima riga. |
| Reverse-continue | Esecuzione fino al al prossimo breakpoint o Watchpoint. |

Undo-DB

I debugger reversibili, nonostante i vantaggi che propongono, non sono ancora molto usati dato che sono poco ottimizzati.

Per questo, **Undosoftware** ha migliorato **GDB** creando:

UndoDB: debugger reversibile più performante, che possiede consumi di overhead di tempo e memoria più bassi.

13. VERSIONING

A. VERSION CONTROL SYSTEM

Sistema che registra le modifiche apportate, ad un file o ad una serie di file, nel tempo, in modo da poter chiamare una versione specifica in seguito.

Viene tenuta traccia:

| Di chi ha eseguito la modifica | Perchè è stata eseguita | Riferimenti a problemi risolti o miglioramenti introdotti |
|--------------------------------|-------------------------|---|
|--------------------------------|-------------------------|---|

Questo permette di avere una panoramica dei cambiamenti e di avere una maggiore coordinazione nel lavoro tra i Teams.

I due sistemi più utilizzati sono:

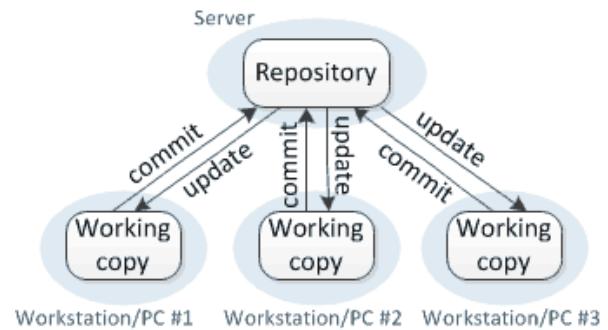
- SVN** —> Sviluppato da Apache Soft Foundation;
- GIT** —> Inizialmente, sviluppato per tenere traccia delle versioni Linux.

SVN :

Utilizza un **modello centralizzato**, infatti tutti hanno una copia locale e le modifiche vengono archiviate in una repository centrale, per questo (per esempio) bisogna essere connessi online per eseguire il commit.

La ramificazione (branch) e la “fusione” (merging) richiedono molto tempo.

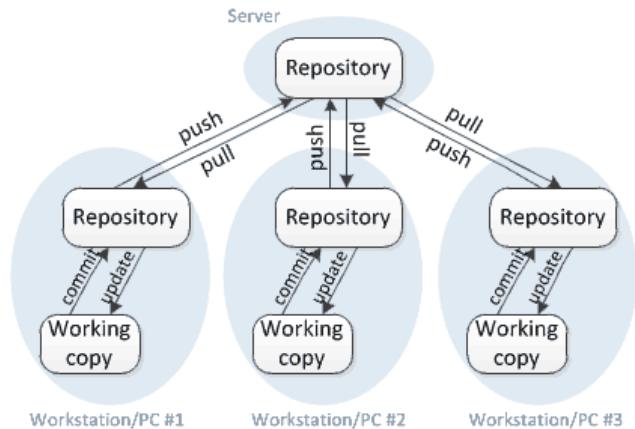
Centralized version control



GIT :

Utilizza un **modello detto “distribuito”**. Ogni utente ha la propria copia del codice in locale e continua ad apportare le proprie modifiche fino a quando non è soddisfatto, per poi successivamente unirlo al proprio “master”, il quale è in locale.

Distributed version control



LOCALE (GIT) VS CENTRALIZZATO (SVN) :

| Locale | Centralizzato |
|--|---|
| Ogni utente ottiene il proprio repository e la propria copia di lavoro. Quindi, eseguendo un commit, i colleghi non avranno accesso all'aggiornamento delle tue modifiche, fin quando non le si invieranno alla repository centrale. Oltre a ciò, non otterremo le modifiche altrui fin quando non le inseriremo nel nostro repository. | Ogni utente ottiene la propria copia di lavoro, ma esiste solo una repository centrale. Quindi, quando si esegue un commit, i colleghi, quando aggiornano, vedono subito le modifiche. |

COMMANDI DI BASE SVN E STRUTTURA:

Il comando:

```
Svn checkout
```

Viene usato per **creare una copia di lavoro del progetto.**

Tale operazione deve essere eseguita ogni volta dopo ogni modifica apportata sulla directory, la quale, se viene modificata, potrebbe essere necessario ricontrollarla.

Questo commando viene eseguito come:

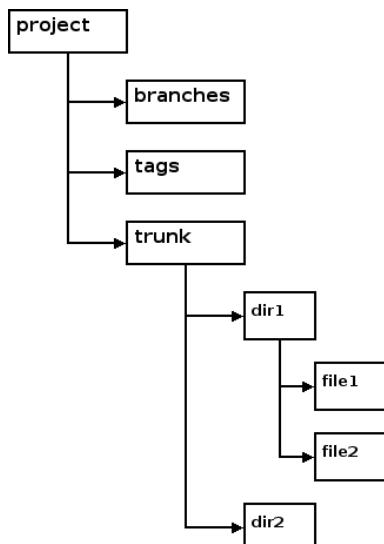
```
svn checkout URL PATH  
svn ci URL PATH
```

Invece, il **commit svn viene usato per salvare le modifiche apportate sulla repository**, ovvero ogni volta che apportiamo modifiche sulle nostra copia di lavoro e vogliamo rifletterle sul server.

Questo viene eseguito come:

```
Svn commit -m "Messaggio per il commit"
```

La struttura visiva di tale modello è:



Dove, possiamo notare, **che il ramo principale di lavoro è il “Trunk” e la ramificazione può essere usata per provare e sviluppare nuove funzionalità**, senza influenzare il truck con eventuale errori o/e bug.

Una volta che la nuova funzionalità è stata completata, il nuovo ramo può essere integrato nel Trunk.

TAGGING:

Grazie a queste etichette è **possibile contrassegnare determinate versioni e ricreare la build in un secondo momento**, infatti possiamo immaginare ciò come una creazione di una “istantanea” del progetto in una determinata fase.

La creazione di un Tag avviene come la creazione di un branch, con la differenza che il tag si troverà all'interno della directory dei tag.

In SVN non c'è differenza tra rami e tag, ma la differenza viene data dall'uso che ne fa l'utente con la directory.

Possiamo però differenziarli dal fatto che, generalmente:

| Rami | Tag |
|--|--|
| Vengono creati, modificati e riuniti nel truck. | Vengono creati come instantanea del progetto in un determinato momento, quindi non vengono mai modificati . |

SVN DEPRECATED:

Ad oggi, il metodo SVN è sempre meno usato, per colpa della sua struttura centralizzata.

Infatti, **GIT ormai è diventato il nuovo modello standard per il controllo delle versioni del codice**, anche perché può essere facilmente integrato con **strumenti di sviluppo in una toolchain** (ovvero, sequenza di strumenti utilizzati per creare un prodotto software) fluida.

COMANDI BASICI DI GIT

Premettendo che con “Area di staging” intendiamo un'area di lavoro virtuale che funge da punto di raccolta temporaneo per le modifiche che si desidera includere nel prossimo commit.

Per iniziare, visioniamo **come poter ottenere un progetto git**, tramite due “strade”:

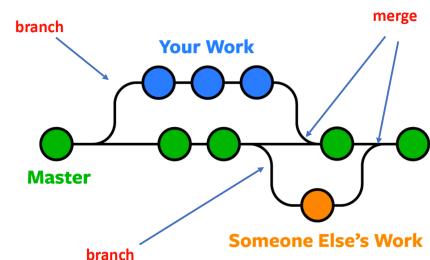
| | |
|----------------------|--|
| git.init *.c | “Prende” un progetto / directory e lo importa in GIT. |
| git commit -m | Clona una repository GIT esistente da un altro server. |
| Git add | Usato per aggiungere un file che si trova nella directory di lavoro all'area di “stagni”. |
| Git commit | Usato per aggiungere tutti i file in fase di stage al repository locale. |
| Git push | Usato per aggiungere tutti i file impegnati nel repository locale a quello remoto. Quindi, in quello remoto, tutti i file e le modifiche saranno visibili a chiunque abbia accesso al repository remoto. |
| Git fetch | Usato per trasferire i file dal repository locale nella directory di lavoro. |
| Git merge | Usato per trasferire i file dal repository locale nella directory di lavoro. |
| Git pull | Usato per ottenere files da una repository remota direttamente nella directory di lavoro. (Equivalente ad un git fetch / git merge). |

Inoltre, evidenziamo due “figure”:

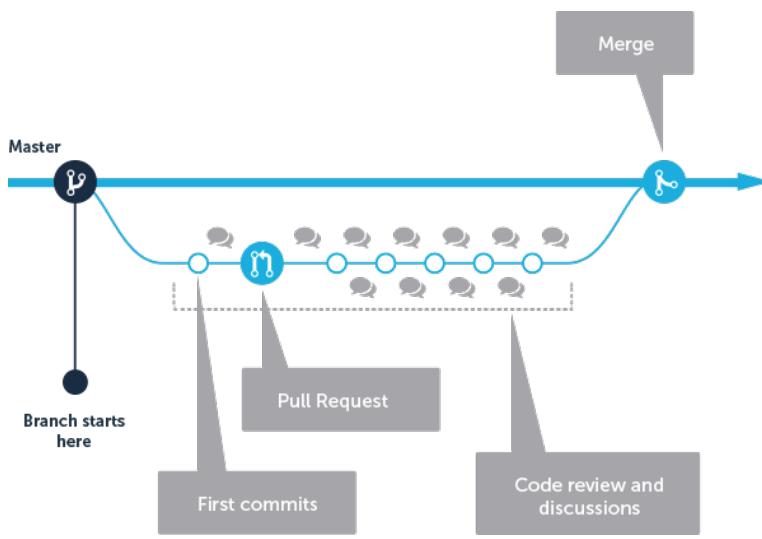
| Master | Head |
|--|---|
| Ramo predefinito, generato quando è stata inizializzata una repository in git. | Puntatore / etichetta applicata al commit più recente del ramo in cui si trova attualmente. |

Altri comandi per i rami:

| | |
|--|--|
| Git branch | Elenca tutti i rami disponibili. |
| Git branch <nome> | Crea un nuovo ramo. |
| Git switch <ramo esistente> | Permette di cambiare ramo in cui lavorare. |



PULL REQUEST:



Funzionalità che permette di semplificare la collaborazione in git tra i vari sviluppatori, sono un meccanismo che permette di notificare ai membri del team che si è completata una funzionalità.

Infatti, quando un ramo che contiene una funzionalità pronta, lo sviluppatore invia una **pull request**, consentendo così ai soggetti coinvolti di sapere che devono rivedere il codice e unirlo al ramo principale.

FAILED AUTOMATICHE MARGE:

| Git aggiunge automaticamente tre indicatori accanto alle righe del codice che è in conflitto:

| | |
|---------|--|
| “<<<<<“ | Seguito da “HEAD” che si riferisce al ramo corrente, indicano che l'inizio delle modifiche è all'interno della sezione indicata. |
| “=====“ | Indicano la fine delle revisioni all'interno del ramo corrente e l'inizio in un ramo nuovo. |
| “>>>>>“ | Seguito dal ramo in cui è avvenuto un merge, indicano la fine delle modifiche all'interno del ramo in conflitto. |

B. SEMANTIC SW VERSIONING

Il controllo delle versioni del software è il **processo di assegnazione di nomi di versioni univoche / numeri di versioni univoche a stati univoci del software** del computer.

Per spiegarci meglio:

All'interno di una determinata categoria di numeri di versione (possiamo far riferimento a Major o Minor) e, generalmente, sono assegnati in ordine crescente in base ai nuovi sviluppi del software.

Prendiamo esempio i software dei computer, che vengono monitorati usando due schemi di controllo delle versioni software:

| Numero di versione interno | Versione di rilascio |
|---|---|
| Può essere incrementato diverse volte al giorno, è come un numero di controllo delle revisioni . | Cambia meno spesso, dato che è come un nome in codice del progetto . |

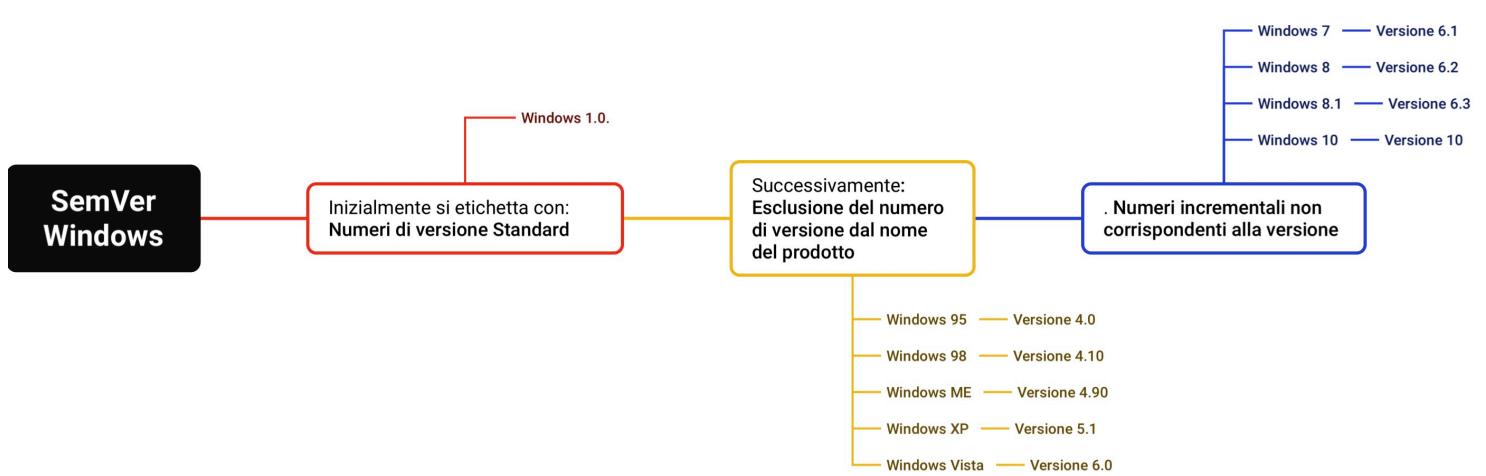
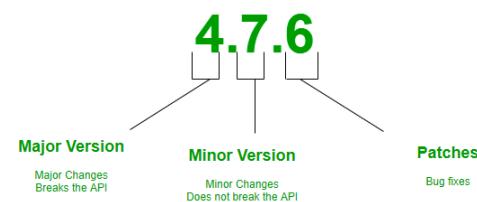
VERSIONING SEMANTICO

Noto anche come SemVer, è un sistema che è andato aumentato negli ultimi anni.

Questo, perchè, in generale, **avere un modo universale di controllare le versioni dei progetti è il modo migliore per tenere traccia del percorso del software**, dato che ogni giorni vengono creati nuovi plug-in, componenti aggiuntivi, librerie ed estensioni.

Questo Versioning, è **composto da un numero a 3 componenti**, dove:

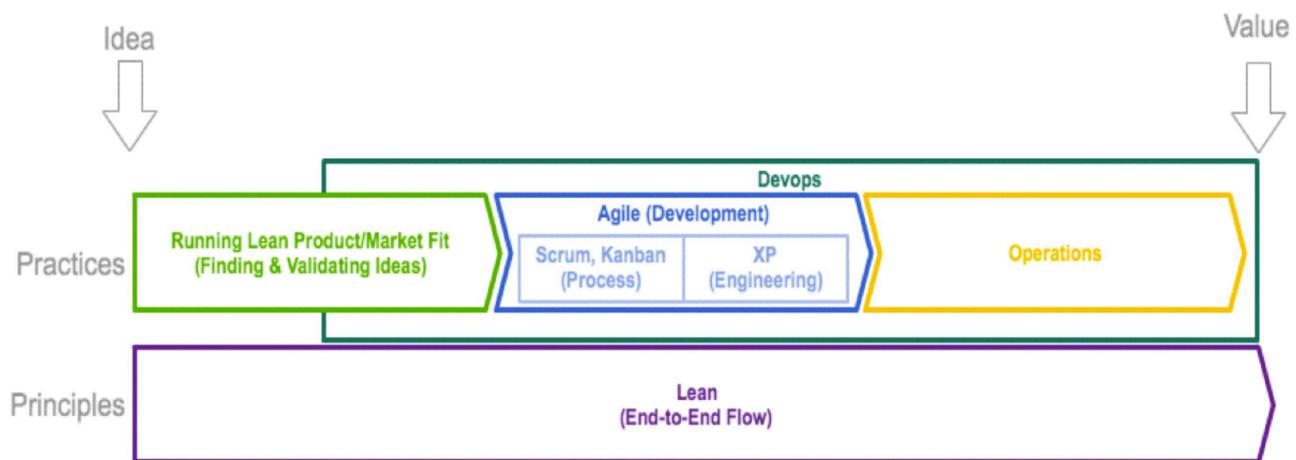
| Cifra a Sinistra | Cifra al Centro | Cifra a Destra |
|--|--|-------------------------------------|
| Sta per il numero della versione principale . | Sta per il numero versione minore . | Sta per il numero di patch . |



14. DEVOPS

Pratica dove, gli ingegneri che si occupano di development e di operations partecipano insieme nell'intero ciclo del servizio: dalla fase di design, al processo di sviluppo fino allo sviluppo della produzione.

Tale pratica è l'applicazione della metodologia Agile nell'amministrazione di sistemi.



Vantaggi

Le organizzazioni IT rilasciano più frequentemente, in tempi più brevi, **falliscono di meno** e sono capaci di riprendersi dagli errori più velocemente.

L'aspetto di management è più semplice e le consegne continue permettono di consegnare "valore" più velocemente.

Elevate performance sono raggiungibili, anche se le app siano:

- Greenfield (ovvero, senza vincoli legati ad una architettura Legacy soft (obsoleta));
- Brownfield (ovvero, sviluppo software che avviene su sistemi o basi di codice esistenti).

PRINCIPI DEL DEVOPS - C.A.M.S.

Codesti principi sono riassumibili nell'acronimo C.A.M.S. e sono i seguenti:

• Culture :

Definita dalle interazioni tra le persone e i gruppi.

Mentre il modello tradizionale prevede la separazione del gruppo dei developers e operations, questo modello prevede di cambiare tale cultura e renderli uniti tenendoli sulla stessa "pagina" e, quindi, facendogli condividere anche le responsabilità.

• Automation:

Tale principio può far risparmiare tempo, effort (fatica) e soldi.

Anche l'Automation, come la culture, si focalizza sulle persone e sui processi, non solo sugli strumenti, anche perchè l'impatto della implementazione comprende l'uso di: Integrazione Continua (CI) e Consegna continua (CD), che possono essere ottimizzati dopo aver compreso la culture e gli obiettivi dell'organizzazione.

Quindi è utile pensare ciò come un “acceleratore” che potenzia i vantaggi del DevOps.

• Measurements:

Le misurazioni aiutano a determinare se si stanno facendo progressi nella “direzione” prevista.

Bisogna avere due accortezze sulle metriche, ovvero:

1. Assicurarsi che le metriche che usiamo siano quelle corrette;
2. Incentivare l'uso delle metriche giuste.

Le principali metriche inclusi, ad esempio:

| | | | | |
|----------|-------|-----------------------------|------------------------------|----------|
| Guadagni | Costi | Tempi medi tra i fallimenti | Soddisfazione dei dipendenti | E altri. |
|----------|-------|-----------------------------|------------------------------|----------|

Queste pratiche, classificate come DevOps, incoraggiano ad avere una visione come **“Alberi in una foresta”**, quindi ad osservare l'operazione intera nell'intero complesso.

• Sharing:

I processi DevOps offrono una grande trasparenza e apertura.

Infatti, diffondere la conoscenza aiuta ad allargare i loop di feedback, permettendo così all'organizzazione di migliorarsi di continuo.

Questa intelligenza collettiva trasforma il team in un'unica unità efficiente, che vale più della somma delle singole parti.

I 3 PILASTRI DEL DEVOPS

| | | |
|-----------------------------------|--|--------------------------|
| Automazione delle infrastrutture. | Continuous Delivery (Consegna continua) | Reliability Engineering. |
|-----------------------------------|--|--------------------------|

Quindi possiamo riassumere il flusso dello sviluppatore in DevOps, come:

| | |
|----|---|
| 1. | Scrivere il codice. |
| 2. | Validare il codice. |
| 3. | Eseguire unit-test sul codice. |
| 4. | Costruire un “Artefatto” (ovvero: trasformare il codice sorgente in un pacchetto distribuibile o file eseguibile) |
| 5. | Consegnare l'artefatto per essere testato. |
| 6. | Eseguire sull'artefatto testato un test di integrazione. (ovvero: verificare che l'applicazione o il servizio funzioni correttamente dopo l'integrazione di nuove modifiche del codice.) |
| 7. | Consegnare l'artefatto alla produzione. |

DELIVERY CONTINUO

Tale viene calcolata tramite due fattori:

$$Delivery\ Continuo = CI + CD$$

Dove:

- **CI = Integration Continua:** Build e creati test.
- **CD = Deployment Continuo:** deployment e test d'integrazione.

Pipeline del Deployment Continuo (CD)



Qui sopra vediamo un esempio di tale Pipeline e deve seguire queste linee guida:

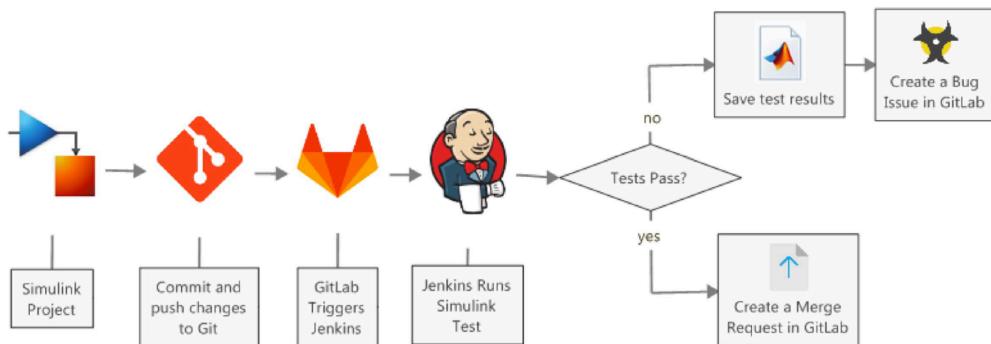
- Gli **artefatti** devono essere **costruiti solo una volta**;
- Gli artefatti devono **essere immutabili**;
- Alla **produzione** deve essere **consegnata** una **copia** prima di **procedere** con la **produzione**;
- Il **deployment** deve **essere fermato se** uno degli **step precedenti fallisce**;
- Le **distribuzioni** devono essere **idempotenti** (*ovvero, non ripetersi*).

Come devono essere per essere considerate come "Migliori Pratiche":

Per essere considerati tali, queste pratiche devono:

- | | |
|----|--|
| 1. | Le build devono passare il “ Coffee Test ” (ovvero, eseguirla in meno di 5 minuti) |
| 2. | Devono essere eseguiti i commit in “ piccoli pezzi ”. |
| 3. | La build non deve essere MAI lasciata rotta . |
| 4. | Non permettere i Flaky Test , ovvero test instabili, ma sistemarli, dato che alla fine della build dobbiamo avere uno status / log / artefatto. |

TOOLS DI DEVOPS



Qui sopra abbiamo un esempio di utilizzo di questi Tools (**Version control e CI System**), che si distinguono come:

| Tools | Applicativo |
|---|--|
| Version Control | Git e altri |
| CI System (Sistemi di integrazione continua) | Jenkins, Bamboo, circleCI, travisCI... |
| Build | Tale dipende dalla piattaforma |
| Test | *unit, robot / protractor, cucumber |
| Artifact Repository | Artifactory, Nexus, DockerHub, S3 |
| Deployment | Rundeck, Ansible |

ACCORGIMENTI NELL’USARE DEL NELL’OPS:

- **Non eseguire task al posto di altri**, ma costruire Tools in modo che ognuno possa eseguire autonomamente i propri task.
- Scrivere codice ed usare pratiche da programmatore per **automatizzare il più possibile i processi**.
- **Costruire percorsi di feedback** che vadano dalla produzione allo sviluppatore.
- Se la produzione necessita, **i developers devono essere disponibili per supporto**.

15. ARCHITETTURE SOFTWARE

Si occupa della progettazione della struttura di alto livello del software, quindi facciamo riferimento all'assemblamento di elementi architetturali per soddisfare determinati requisiti.

L'architettura del software è data da:

$$AS = Elementi + Forme + Limiti$$

Dove:

| | |
|-----------------|---|
| Elementi | Sono i componenti base che costituiscono l'architettura , come moduli / pacchetti / classi.. |
| Forme | Definiscono la struttura modulare del software e come gli elementi si interconnettono tra di loro. Quindi definiscono le relazioni e come possono interagire e scambiare dati . |
| Limiti | Stabiliscono vincoli / restrizioni che “governano” il comportamento del software. In sostanza, definiscono cosa o non può fare e come deve interagire il software . |

Differenziamo:

| Architettura | Design |
|--|---|
| Riguarda le decisioni globali, quindi a livello del sistema, sottocomponenti e le loro dipendenze. | Riguarda le decisioni locali, quindi a livello interno. |

ACCOPPIAMENTO E COESIONE

Gli elementi basici dell'Architettura del Software sono i moduli e connettori.

Si dice che si ha una buona architettura quando si:

| Minimizza l'accoppiamento tra i moduli | Massimizza la coesione all'interno di un modulo |
|--|---|
| Cercando di avere meno interconnessioni possibili, facilitando modifiche future. | Ovvero, tutte le funzioni di ogni modulo devono essere ben correlate, permettendo così maggior comprensione dell'architettura software. |

PER CHI È UTILE L'ARCHITETTURA DEL SOFTWARE?

1. Project Manager:

Il quale, attraverso la **strategia “divide-et-impera”** può:

| | | | | |
|--|--|--|--|--|
| Assegnare il budget e l' effort (impegno) ai componenti. | Distribuire il lavoro e coordinare il personale. | Assegnare i lavori alle persone più competenti per i vari componenti. | Tracciare il progresso del progetto, basandosi sui singoli componenti. | Sviluppare un modello di business o strategia di marketing. |
|--|--|--|--|--|

2. Sviluppatori:

I quali dovrebbero implementare i componenti individuali basandosi sull'architettura, la quale deve esser prima compresa prima di far modifiche sull'implementazione.

3. Tester:

I quali devono definire le strategie per il testing, basandosi sull'architettura.

In generale:

Nella costruzione di sistemi è utile per capire come compilare e produrre l'eseguibile finale, come per elementi come Makefile.

CONNETTORI:

Sono qualsiasi meccanismo che permetta a più moduli di interagire, come:

| | | | | | |
|----------------------------|-------|------|------------------|--------|---------------|
| Procedure “Call Remote” | Files | Pipe | Code di messaggi | Socket | Shared Memory |
|----------------------------|-------|------|------------------|--------|---------------|

Moduli VS Connettori:

| Moduli | Connettori |
|--------------------------------------|---|
| Forniscono funzionalità applicative. | Specificano il protocollo di interazione tra componenti e sono , solitamente, indipendenti dal dominio applicativo . <i>Sono implementati dai linguaggi di programmazione o dal Sistema Operativo.</i> |

Utilizzo dei Connitori:

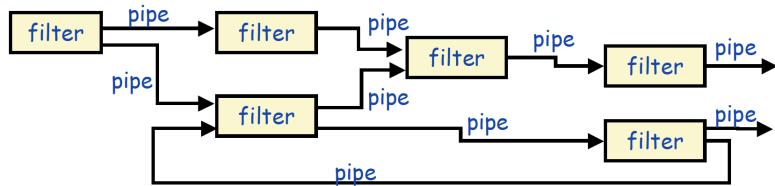
Possono essere dotati di una logica e possono assumere diversi ruoli:

| Coordinator | Wrapper | Adattatore | Monitor |
|---|--|--|--|
| Fa da “tramite” tra componenti diversi. | Nasconde alcune funzionalità di un componente. | Fa da “mediatore” tra componenti che agiscono su file di diverso tipo. | Monitora i comportamenti dei componenti. |

STILI ARCHITETTURALI

Pipe and Filter:

In questo modello **si usano principalmente le pipe**: "tubi" che hanno una sola sorgente e più diramazioni.



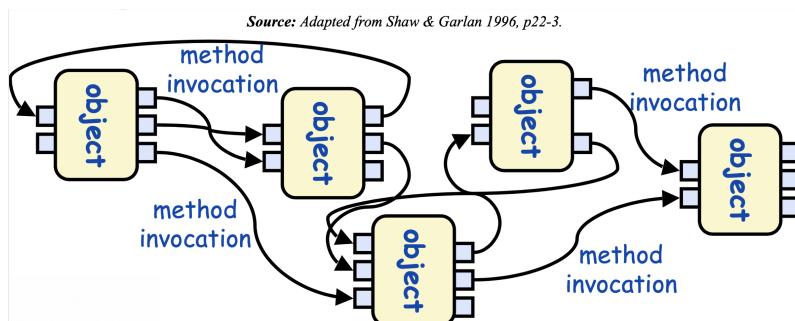
Come da foto, si hanno più filtri in parallelo che vengono organizzati con le pipe.

| | |
|-----------|--|
| Proprietà | I filtri <u>non</u> hanno bisogno di conoscere nulla dei componenti ai quali sono collegati. |
| | I filtri <u>possono</u> essere implementati in parallelo. |
| | Il comportamento globale del sistema <u>dipende</u> dalla composizione dei comportamenti dei singoli filtri. |

Object Oriented:

In questa architettura:

- I moduli sono oggetti;
- I connettori sono invocazione ai metodi.

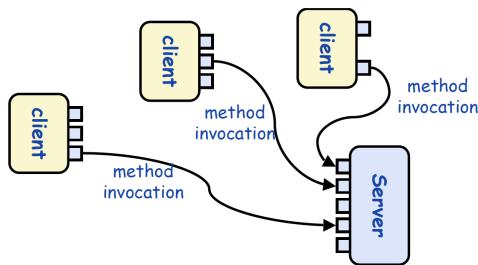


| Vantaggi | Svantaggi |
|--|--|
| Offre il data-Hiding (Rappresentazione interna può essere nascosta al cliente) | Gli oggetti devono conoscere a priori con quali oggetti interagire . |
| È facilmente scomponibile . | Ha problematiche con i sistemi distribuiti . |
| Può essere multi-threading | |

Ha 2 specializzazioni:

1. Client-Server:

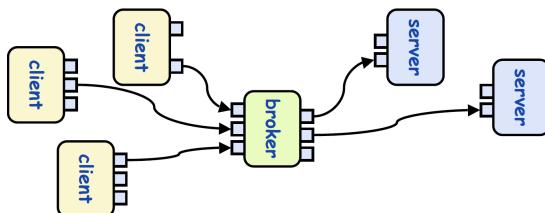
I cui sono sempre collegati da invocazioni a metodi:



| Vantaggi | Svantaggi |
|--|---|
| Un client <u>non</u> ha bisogno di conoscere gli altri client. | Un client <u>deve</u> conoscere il server. |
| | Se il <u>server cade</u> , il <u>client cessa</u> . |

2. Broxy / Proxy:

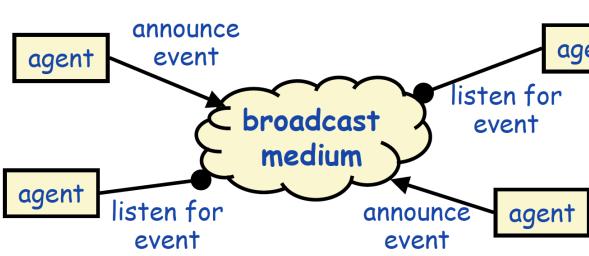
Dove i **Broxy / Proxy** fa da intermediario tra **client** e **server**, così i client non necessitano di conoscere i server o viceversa.



| Vantaggi | Svantaggi |
|--|---|
| I client <u>non necessitano</u> di conoscere quale server usare. | I broker diventano “Colli di bottiglia” (Punti di congestione all'interno dell'infrastruttura di rete che causano un rallentamento / interruzione del flusso di traffico.) |
| I client può avere più broker, quindi anche più server. | Le performance sono peggiori. |

Event Based (Invocazione Implicita)

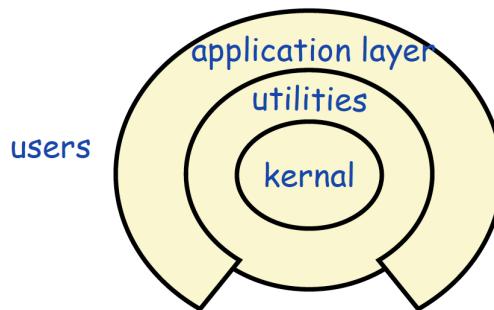
Si definisce evento un cambiamento di stato, dove alcuni agenti li annunciano e altri si registrano a questi.



| Vantaggi | Svantaggi |
|---|---|
| Chi produce gli eventi, non ha bisogno di conoscere chi è registrato. | Non è presente un controllo dell'ordine degli eventi. |
| Supporta il riuso e l'evoluzione del sistema, dato che si può aggiungere eventi facilmente. | |

A Layer:

Dove si sviluppa prima un “core” su cui vengono poi applicati altri moduli, come avviene nei sistemi operativi.



| Vantaggi | Svantaggi |
|--|---|
| Supporta l'incremento del livello di astrazione. | In alcune situazioni, è difficile individuare bene i layer e quindi non è utilizzabile. |
| Supporta l'uso e l'evoluzione. | |

Può essere di due tipi:

| Chiusa | Aperta |
|---|--|
| Ogni layer può comunicare <u>solo col layer sottostante</u> . | Ogni layer può comunicare <u>con tutti i layer sottostanti</u> . |

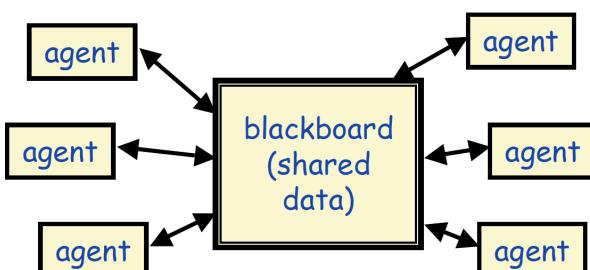
Istanze di architetture a layer:

- 2° Layer: Client e Database;
- 3° Layer: Servono per separare il livello di interfaccia utente dalla business logic e dal database.
- 4° Layer: Separa le applicazioni dalle entità che usano.

Architettura basata sulle repository:

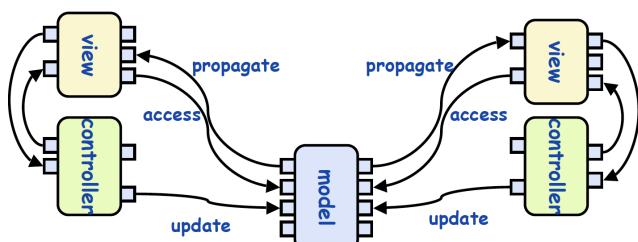
Simile a quella event based.

Presenta una lavagna su cui tutti gli agenti possono scrivere e leggere.



| Vantaggi | Svantaggi |
|--|---|
| Si evita di duplicare dati complessi. | La lavagna diventa un “collo di bottiglia”. |
| Tempo e spazio sono dissociati, anche se un agente termina i suoi dati rimangono presenti. | |

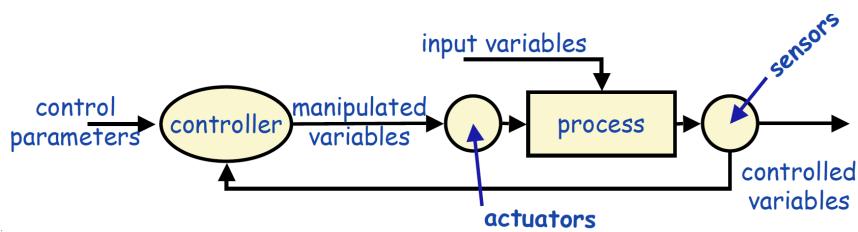
Una specializzazione è l'architettura Model View control:



| | | |
|---|--|---|
| Abbiamo un unico modello con diverse vie. | Ogni vista ha un controllo associato che fa da “Broker”. | I cambiamenti vengono applicati a tutte le liste. |
|---|--|---|

Control System:

Dove un sensore manda segnali al controller:

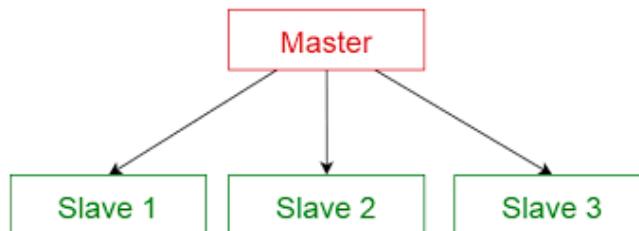


| Vantaggi | Svantaggi |
|--------------------------------------|--|
| Utilizzato per i sistemi “Real-Time” | Difficile da specificare le caratteristiche temporali. |

Master / Slave:

È analoga al client server, in quanto è presente un master e uno slave.

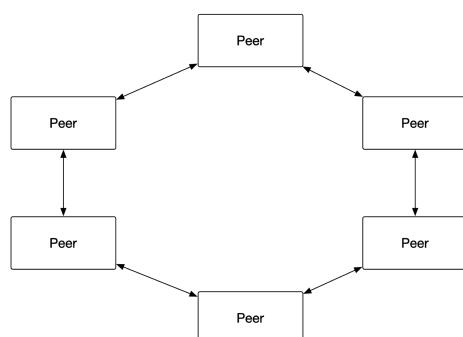
Viene usata quando si ha un problema complesso che si può scomporre in sotto-problemi uguali.



| Vantaggi | Svantaggi |
|---|---|
| Il master scomponete il problema in sotto problemi identici e li delega agli “Slave”. | Il problema deve essere divisibile in sottoproblemi identici. |

Peer-2-Peer

Usato nei sistemi distribuiti / condivisi e nelle blockchain.



| Vantaggi | Svantaggi |
|-----------------------------|---|
| Tutti i moduli sono “Pari”. | Non è presente un controllo centrale che supervisiona il sistema. |
| | Un peer potrebbe controllarsi in maniera malevola. |
| | Necessità di strategie di incentivazione. |

16. LICENZE SOFTWARE

CHE COS'È?

È un documento che **definisce linee guida**, a livello giuridico, per l'utilizzo e la distribuzione del software.

I termini e le condizioni di una licenza software, **soltamente, includono:**

| | | | |
|--------------------------------------|---|--|--|
| Uso corretto del software. | Limitazioni di responsabilità. | Garanzie ed esclusioni di responsabilità. | Definiscono le protezioni , in caso il software o il suo utilizzo violino la proprietà intellettuale altrui |
|--------------------------------------|---|--|--|

E le licenze possono essere:

| | | |
|--------------|----------|-------------|
| Proprietarie | Gratuite | Open source |
|--------------|----------|-------------|

Dove si va a distinguere i termini in base ai quali gli utenti possono ridistribuire / copiare il software per sviluppo o utilizzo futuro.

COME FUNZIONANO?

Viene **definito** come il **software** può **essere utilizzato** e **come verrà pagato**.

In queste, poi, può essere inclusi anche:

| | | |
|--|----------------------------|--|
| Quante volte il software può essere scaricato | Quanto costerà il software | Quale livello di accesso gli utenti al codice sorgente. |
|--|----------------------------|--|

Essendo un contratto tra utente / organizzazione e lo sviluppatore, la definizione dei termini di licenze, deve essere accettata dall'utente al momento dell'acquisto del software, dove verrà fornita una **"License Key"** / **"Product Key"** che verrà utilizzata per identificare e verificare la versione specifica del software e anche per far attivare il software su un dispositivo.

PERCHÈ IL LORO UTILIZZO?

Perchè si riesce così a **stabilire i diritti di tutte le parti coinvolte nel software**:

| | | |
|---------|------------|----------------|
| Autore. | Fornitore. | Utenti finali. |
|---------|------------|----------------|

Definendo il rapporto tra la società di software ed utenti e **spiegando come vengono "protetti"**.

COME VENGONO PROTETTI GLI SVILUPPATORI?

Grazie alla licenza, si riesce a:

| |
|---|
| A. Proteggere la proprietà intellettuale ed i segreti commerciali degli sviluppatori , seguendo le leggi sul diritto d'autore. |
| B. Limitare ciò che gli altri possono fare col codice software. |
| C. Limitare la responsabilità del venditore. |

COME VENGONO PROTETTI GLI UTENTI?

I contratti di licenza permettono di proteggere gli utenti:

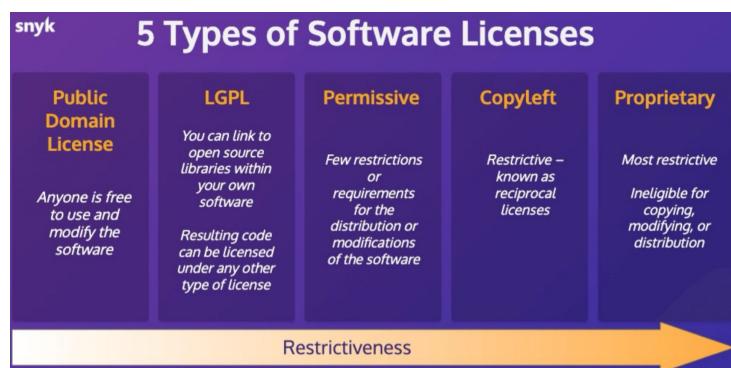
1. **Definendo cosa possono fare** gli utenti **col codice** software.
2. **Stabilendo in che modo** gli utenti **devono rispettare la licenza** software, proteggendosi così dalle denunce di violazione e limitando la loro responsabilità legale.
3. Aiutando gli utenti a **mantenere un rapporto “positivo” con gli sviluppatori** e fornitori del software.
4. **Evitando spese excessive per le licenze**, stabilendo quante licenze sono necessarie per un’organizzazione.

EULA: CONTRATTO DI LICENZA CON L’UTENTE FINALE:

Una delle caratteristiche delle licenze è che **l’editore del software concede l’uso di una / più copie del software, in base al contratto di licenza stipulato con l’utente**, ma la proprietà di tali copie rimane all’editore del software.

Quindi, è **tipico degli EULA includere termini che definiscono gli usi del software**, come il numero di installazioni consentire o i termini di distribuzione.

CLASSIFICAZIONE PIÙ SELETTIVA DELLE LICENZE:



Public Domain License:

Con tale licenza, **chiunque è libero di utilizzare e modificare il software senza restrizioni**.

Si tratta di una **licenza “permissiva”** che **consente di adottare il codice in applicazioni / progetti e di riutilizzare il software come si desidera**.

Le aziende devono prestare attenzione quando adottano tali software per i propri progetti, dato che:

- Potrebbero non sempre aderire alle migliori pratiche di codifica o non essere allaltezza.
- Il software che non rientra in termini di licenza specifici non è sempre codice di pubblico dominio.

LGPL (o GNU) - GNU LESSER GENERAL PUBLIC LICENZE

Licenza di software libero flessibile e ampiamente utilizzata che consente l'utilizzo di librerie open source in progetti software proprietari, promuovendo la collaborazione e l'innovazione pur proteggendo la proprietà intellettuale.

! Descrizione data da Gemini.

LGPL o GPL?

| | |
|------|---|
| LGPL | Più permissiva e consente l'utilizzo del codice in software proprietario , purchè l'interfaccia della libreria rimanga aperta. |
| GPL | Offre la massima libertà per la modifica e la ridistribuzione del software , ma richiede che tutto il codice derivato sia open source. |

LGPL e biblioteche:

Tramite questa licenza, **gli altri sviluppatori possono prendere "in prestito" e modificare il codice e ri-distribuirlo come parte del proprio progetto, a condizione che la parte utilizzata sia sotto questa licenza**, comprese anche le eventuali modifiche.

Bisogna considerare che:

- Quando si utilizza codice LGPL in software proprietario, **l'interfaccia della libreria deve rimanere aperta e accessibile**, ma il codice sorgente interno può essere protetto.
- Il codice LGPL **può essere collegato dinamicamente a software proprietario, ma non può essere collegato staticamente**.

Permissive:

È una delle **più comuni e popolari** tra le licenze software open source, dato anche dal fatto che **sono presenti poche restrizioni o requisiti per la distribuzione o la modifica** del software.

Sono **presenti varie varianti** di questa tipologia, **come la licenza MIT, che includono differenze nei requisiti**, con lo scopo di preservare gli avvisi di licenza ed i diritti di autore per il software (come il software può essere utilizzato, requisiti di marchio e altro).

CopyLeft:

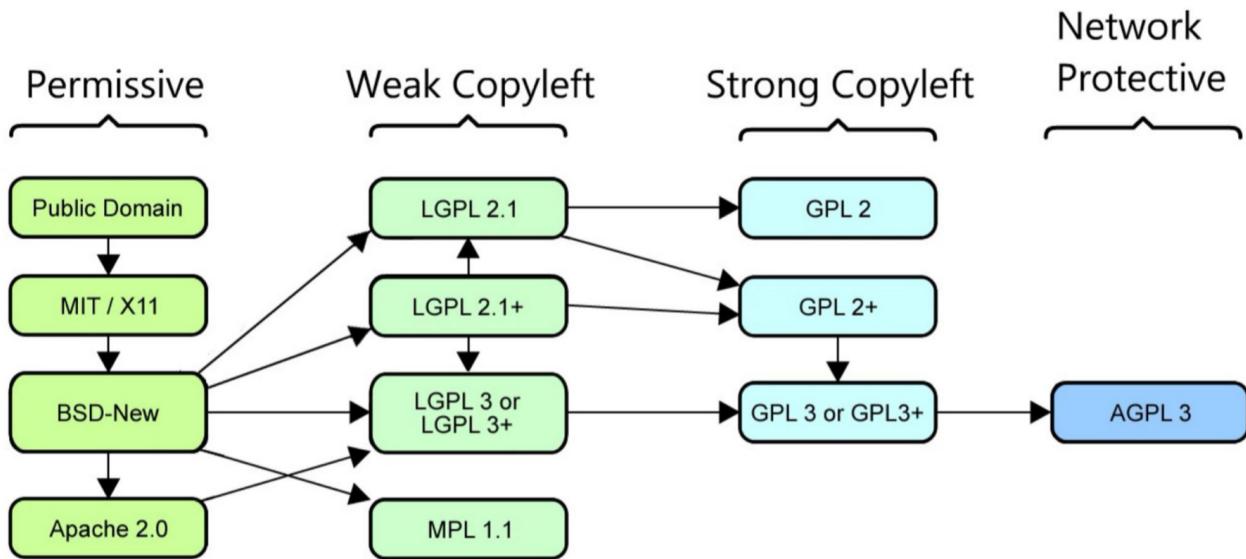
I **termini** di questa licenza sono **restrittivi**, infatti il **codice concesso** in licenza **può essere modificato o distribuito come parte** di un **progetto** software, solo **se il nuovo codice è distribuito con la stessa licenza**.

Quindi, se il codice incluso nel prodotto software è stato specificato per essere per "uso personale", il nuovo prodotto deve portare la stessa restrizione.

Proprietario:

Questo è il tipo più **restrittivo** di licenza software, **infatti rendono il software non idoneo alla copia / modifica / distribuzione**.

PERMISSIVE, COPYLEFT AND OTHER:



Le licenze open source possono suddividersi in due categorie principali:

| CopyLeft | Permissive |
|----------|------------|
|----------|------------|

Dove le differenze sono:

| Persistenza | Effetto Virale |
|---|--|
| Tutte le opere derivate devono essere concesse in licenza e distribuite con la stessa licenza d'origine | La combinazione di un'opera copyleft concessa con una licenza diversa, fa sì che la "fusione" restituisca una opera sotto licenza Copyleft |
| Le licenze Copyleft lo sono. | Le licenze Permissive non lo sono. |
| | Licenze Copyleft hanno tale effetto. Licenze Permissive non hanno tale effetto. |

Oltre a ciò, le licenze Copyleft si distinguono in:

| Forte | Debole |
|-------|--------|
|-------|--------|

Quando **non viene specificata**, si deve assumere come "**Forte**".

Entrambe le licenze sono persistenti, ma **solo le forti hanno l'effetto virale**.

Quindi, **le opere concesse in licenza con CopyLeft debole, possono essere incluse in un'opera aggregata concessa in licenza usando una licenza diversa**, questo con la condizione che deve essere disponibile agli utenti, anche quando è incluso un'opera aggregata che è concessa con altra licenza.