1. Processi software

Secondo l'ISO, un **processo** è un insieme di attività correlate e coese che trasformano ingressi in uscite consumando risorse. Importante ricordare che un processo software (d'ora in poi solo *processo*) porta ad un *prodotto* software. Ogni processo si divide in **attività** ed ognuna di queste è a sua volta divisa in altre parti dette **compiti**. Ogni compito si svolge con certe tecniche scelte in base a quello che bisogna fare.

Per avere qualità di un prodotto, l'ingegneria del software mette a disposizione gli standard di processo condivisi tra aziende; queste poi specializzano i processi in processi definiti adattandoli alle proprie esigenze. Per ogni progetto, l'azienda crea dei processi di progetto che usano risorse aziendali per raggiungere gli obiettivi. Ricapitolando:

- Standard di processo — una sorta di "template" usato come riferimento di base generico per lo svolgimento di funzioni aziendali
- Processo definito — specializzazione dello standard fatta per esigenze specifiche del progetto; una azienda può "customizzare" uno standard
- Processo di progetto — applicazione di un *Processo definito* ad un progetto reale che usa risorse aziendali per raggiungere degli obiettivi

Non esiste il processo perfetto in quanto nessun progetto è identico ad un altro; i processi vanno selezionati in modo critico in base a quello che si deve fare. Selezionare, adattare, applicare e migliorare i processi è compito dell'amministratore di progetto.

Lo standard ISO/IEC 12207 è il più noto standard di processo che divide i processi in tre famiglie principali. Descrive le attività ed i compiti per ciascuno e fornisce un modello da specializzare:

- Processi primari — Se c'è un processo software, allora ci deve essere anche un processo primario. Le sue attività principali sono:
    - Fornitura — cioè gestione dei rapporti con i clienti
    - Sviluppo — creazione del progetto
    - Gestione operativa — installazione ed erogazione dei prodotti
    - Manutenzione — correzione, adattamento ed evoluzione

- Processi di supporto — Sono processi che vengono usati da altri processi quando serve e danno supporto a chi li esegue. Le sue attività principali sono:
    - Documentazione
    - Gestione delle versioni e della configurazione
    - Risoluzione di problemi
    - Verifica e validazione (*V&V*)

- Processi organizzativi — Sono i processi impiegati dall'azienda per garantire e gestire il lavoro. Le sue attività principali sono:
    - Formazione del personale
    - Miglioramento del processo
    - Gestione delle infrastrutture
    - Gestione dei processi

Serve una forma di standardizzazione per "tenere alta" la qualità di un processo che rischia continuamente di degradare. Ecco perché un buon processo si auto-migliora di continuo secondo lo schema **PDCA** (il ciclo di Deming):

1. Plan — individuare gli obiettivi di miglioramento
2. Do — mettere in atto ciò che si è deciso al punto precedente

3. Check — verificare che tutto abbia funzionato; studiare i risultati della fase *Do* e confrontarli con gli obiettivi della prima fase *Plan*
4. Act — correggersi; se la fase prima (*Check*) ha mostrato che il piano (*Plan)* implementato (*Do*) è migliore rispetto a prima, allora questo piano diventa il nuovo standard (la nuova **baseline**).

Importante è sottolineare la differenza che c'è fra i due termini *efficacia* ed *efficienza* di un processo:

- Efficace: un processo (o attività o compito) è efficace se raggiunge gli obiettivi attesi
- Efficiente: un processo (o attività o compito) è efficiente se fa quello che deve fare senza sprecare risorse

2. Ciclo di vita del software

Conviene vedere il ciclo di vita di un software come una macchina a stati dove gli stati sono il grado di maturazione del prodotto e gli archi sono le attività che fanno migliorare il prodotto (e che lo portano quindi alla maturazione). Si chiama **fase** il tempo che intercorre fra uno stato ed un altro. La scelta del modello del ciclo di vita va fatta prima di pianificazione e progettazione.

Ci sono tre tipi di manutenzione che si possono fare durante il ciclo di vita:

- ☹ Correttiva — corregge i bug
- 😐 Adattiva — adatta il sistema a variazioni dei requisiti
- 😊 Evolutiva — aggiunge funzionalità al sistema

I modelli del ciclo di vita del software sono molti però i più significativi sono sostanzialmente quattro:

- Modello sequenziale — è ispirato alle catene di montaggio. Si tratta di una successione di **fasi** sequenziali dove si può solo andare in avanti (quindi non tornare a stati precedenti) e non ci si può trovare contemporaneamente in due stati. Il passaggio alla fase successiva è basato sulla documentazione. I cicli sono iterazioni.

  Il difetto è che tutto è molto rigido e bisogna già sapere a monte cosa fare; funziona solo se il cliente sa precisamente quello che vuole; il vantaggio è che si possono individuare fasi distinte e ordinate fin da subito.

- Modello incrementale — qui i cicli non sono più iterazioni ma incrementi; ogni incremento attraversa tutte le fasi del modello sequenziale. Ci sono rilasci multipli che realizzano un incremento di funzionalità che punta ad una approssimazione sempre migliore della soluzione da raggiungere.

  Il vantaggio è che le funzionalità più importanti vengono affrontate all'inizio e così facendo vengono verificate più volte, dato che sono sempre presenti in ogni iterazione. Ogni incremento riduce il rischio di fallimento. Il modello sequenziale segue un approccio **predittivo** (cioè si basa sui piani che devono essere rispettati) mentre il modello incrementale segue un approccio **adattivo** (cioè non si basa su piani già stabiliti ma considera la realtà imprevedibile).

- Modello evolutivo — è incrementale e prevede che gli incrementi successivi siano versioni (*prototipi*) utilizzabili dal cliente. Più versioni possono essere mantenute in parallelo ed ogni fase ammette più iterazioni.

- Modello agile — è un modello altamente dinamico che dà maggiore importanza al software e meno importanza alla documentazione, dà maggiore importanza alle persone e meno importanza ai processi. L'idea di base è l'**user story**, cioè un compito significativo che l'utente vuole fare col

software. Il lavoro viene suddiviso in piccoli incrementi che sono sviluppati indipendentemente in sequenza.

Bisogna essere in grado di mostrare in ogni momento al cliente ciò che è stato fatto e verificare l'avanzamento tramite il progresso reale. Un esempio è lo **SCRUM** dove viene utilizzato un "backlog" (una lista di piccoli task da svolgere) ed una persona esperta, chiamata *product owner*, organizza gli **sprint** (breve periodo in cui i task vengono discussi ed assegnati ai programmatori).

- Modello SEMAT/1 — dice che un ciclo di vita è come un automa perché tutte le azioni che svolgo devono farmi progredire. Ogni attività deve essere un miglioramento e per tenerne traccia utilizzo degli indicatori, tipo dei "*termometri*", che mi mostrano lo stato.

  Ci sono tanti blocchi indicanti le attività e ciascuna di esse ha allegato il "*termometro*" che misura il livello di avanzamento. Ogni indicatore comunque va per conto suo, non sono correlati fra di loro.

3. Gestione di progetto

Un progetto è un insieme ordinato di attività che devono essere portate a termine; tali attività sono stabilite dal modello di ciclo di vita che scelgo. Alcune attività le posso svolgere da solo ma il progetto è **sempre** collaborativo. Sono necessari vari ruoli che vengono assegnati a profili professionali differenti in base alle competenze ed esperienze di chi li svolge.

- Responsabile — punto di riferimento per le comunicazioni con il committente e ha la responsabilità di scelta ed approvazione. Partecipa durante tutta la durata del progetto ed approva i documenti.
- Amministratore — è il responsabile dell'**efficienza** e dell'operatività dell'ambiente di sviluppo. Ha la gestione della configurazione del prodotto, del versionamento e della documentazione. Redige le Norme di Progetto.
- Analista — è la persona che ha una grande conoscenza del sul campo ed è colui che cerca di capire il problema. Redige lo studio di fattibilità e l'analisi dei requisiti.
- Progettista — ha una ampia esperienza professionale e si occupa di cercare una soluzione al problema con vincoli accettabili.
- Programmatore — ha competenze tecniche specifiche e si occupa di implementare la soluzione tramite attività di codifica. Concretizza la soluzione del progettista e non sta a lui inventare e non aggiungere cose nuove.
- Verificatore — verifica il lavoro fatto dai programmatori

La pianificazione di progetto serve a definire le attività e se ne prende cura il responsabile di progetto. Tre strumenti notevoli per la pianificazione di un progetto sono:

- Diagrammi WBS — decompongono le attività in sotto attività. Lo scopo è quello di avere paura dei monoliti e dunque spezzare in piccole parti fattibili in **poco** tempo da una sola persona.

- Diagrammi di Gantt — rappresentano la durata, la sequenzialità ed il parallelismo delle attività ma non sono adatti a gestire le dipendenze tra attività. Si pone nell'asse orizzontale il tempo, data una certa unità di misura (se fisso ore, non considero i minuti ma solo le ore tonde), e nell'asse verticale si mettono le attività

- Diagrammi di PERT — unificano le due tecniche precedenti e sono più complessi di Gantt. Sono ideali per rappresentare le dipendenze temporali fra attività e permettono quindi di ragionare sulle scadenze del progetto. Ogni evento ha una data minima in cui può accadere, una data massima

entro il quale va terminata l'attività ed un tempo di **slack** (di quanto tempo un evento può essere ritardato **senza** influenzare l'andamento del progetto).

La stima dei costi di progetto è a carico del responsabile e ci sono due leggi che influenzano la stima dei costi necessari:

- *Il lavoro si espande per riempire tutto il tempo disponibile*
- *Minore è il prezzo di un servizio, maggiore sarà la quantità richiesta*

Queste leggi mostrano che avere troppo tempo a disposizione può avere un impatto negativo sull'efficacia; per avere un prezzo accettabile con la concorrenza è importante tenere bassi i costi. Per stimare i costi si può usare il modello **CoCoMo** e fa stime di tipo tempo/persona un mesi/persona.

I risultati di un progetto software possono portare costi eccessivi, non rispettare le scadenze o risultare insoddisfacenti. Bisogna fare un'analisi dei rischi e cercare di fare in modo di evitarli o mitigarne gli effetti. I livelli di rischio vanno costantemente aggiornati.

4. Amministrazione di progetto

L'amministratore di progetto fa sì che l'infrastruttura di lavoro sia operante e fa scelte tecnologiche. Si occupa di organizzare e gestire l'ambiente di lavoro ed attua le scelte tecnologiche.

La documentazione è necessaria per rendere un progetto **gestibile** e **ripetibile**; siccome la mente è privata non possiamo sapere cosa il project manager ha in testa e dunque, senza documentazione, siamo finiti. I documenti dunque vanno controllati dall'amministratore e ce ne sono di due tipi:

- Documenti di sviluppo — contengono i diagrammi di progettazione, il codice ed i manuali
- Documenti di gestione del progetto — contengono i documenti di accordo col cliente, i piani di qualità ed i consuntivi

L'amministratore deve anche gestire l'ambiente di lavoro cioè persone, ruoli ed infrastrutture; la qualità dell'ambiente di lavoro influenza la qualità del prodotto finale. Dunque bisogna lavorare in un posto completo, ordinato e ben aggiornato. Esempi di tool che l'amministratore deve reperire sono:

- Gestione di progetto — Instagantt per Gantt ad esempio
- Gestione di documenti — Git per il versionamento e LaTeX per la redazione
- Analisi e progettazione — Strumento di tracciamento dei requisiti
- Codifica — InteliJ IDEA per Java ad esempio o Visual Studio per C#
- Integrazione continua — Jenkins per esempio

Altro compito dell'amministratore è la gestione della **configurazione** e del **versionamento** del prodotto, in modo da poterne gestire le varie parti e tenere traccia delle modifiche. Ogni sistema *deve* essere gestito con il controllo di configurazione e controllo di versionamento. In particolare:

- Configurazione — le parti che compongono un prodotto e il modo in cui queste stanno assieme. Una singola parte (un pezzetto) si chiama **Configuration Item** (CI)
- Versionamento — modalità con cui un sistema mette in sicurezza le baseline e consente di tornare a versioni precedenti. Consente tramite un repository di contenere tutti i CI bi ogni baseline e la loro storia.

Ogni CI deve essere unico, avere un ID e vari dati identificativi come nome, data, autore e registro delle modifiche. Data la complessità di un prodotto software, la gestione della configurazione va automatizzata con strumenti adatti. Due concetti centrali nella gestione della configurazione sono:

- Baseline — è un punto di arrivo tecnico dal quale non si retrocede ed è fatto di tanti CI. Si costruisce con la configurazione e si mantiene con il versionamento; è qualcosa di stabile e sta in un repository.
  Quello che ho fino alla baseline funziona, da qui in poi posso fare modifiche e posso sempre tornare a questo punto in caso di problemi.

- Milestone — è un punto strategico nel tempo e comprende un certo numero di baseline. Ogni milestone deve avere obiettivi specifici, essere incrementale nei contenuti (portare sempre cose nuove) e poter essere misurabile in termini di impegno richiesto.

Nel corso di un progetto è normale che ci sia il bisogno di apportare modifiche (richieste da utenti o sviluppatori). Queste però devono essere sottoposte ad un processo di analisi, decisione e realizzazione; vanno tracciate con sistemi di ticketing o di issue tracking.

Il controllo della versione deve basarsi su un repository, cioè un database centralizzato con tutti i CI di ogni baseline; in questo modo si può lavorare in contemporanea su vecchi e nuovi CI senza sovrascritture accidentali. Il controllo di versione porta a diversi risultati:

- Versione — istanza di un prodotto che è diverso da altri
- Release — istanza di un prodotto reso disponibile a tutti gli utenti

5. Ingegneria dei requisiti

Un requisito è un bisogno da soddisfare o un vincolo da rispettare. In particolare un requisito è una caratteristica che un software deve avere e deve essere messo a contratto con il cliente. Ci sono diverse tipologie di requisiti:

- Funzionali — i servizi che il sistema deve fornire e possono essere validati (ad esempio con i test)
- Prestazionali — descrivono il grado di prestazione che deve raggiungere il sistema in certe situazioni (ad esempio la latenza col server)
- Qualitativi — le qualità del prodotto
- Vincolo — i vincoli imposti dal cliente o dal problema e derivano ad esempio da necessità realizzative oppure da obblighi contrattuali

È bene suddividere i requisiti secondo il grado di importanza: *obbligatori*, *desiderabili* o *facoltativi*. Il **Piano di qualifica** garantisce il rispetto dei requisiti definendo le strategie di *verifica* e scegliendo le tecniche per la *validazione*. I due termini sono importanti:

- Verifica — accertare che l'esecuzione delle attività di processo non abbia introdotto errori (*bug*)
- Validazione — assicurarsi che il prodotto realizzato soddisfi i requisiti del committente. In altre parole, assicurarsi che il prodotto realizzato corrisponda alle attese

La validazione **NON** deve fallire e per essere sicuri che ciò non accada esiste la verifica, la quale assicura che va tutto bene *mentre* stiamo lavorando e non *dopo* che abbiamo lavorato. Il processo di *Ingegneria dei Requisiti* raggruppa 4 attività:

1. Studio di fattibilità — studio senza ricerche impegnative svolto all'inizio che valuta i rischi, costi e benefici del sistema in esame. Produce l'omonimo documento e deve tenere conto della fattibilità tecnica e delle scadenze temporali

2. Acquisizione ed analisi dei requisiti — gli analisti devono individuare i requisiti con il proponente comunicando con esso; da una buona analisi dipende la soddisfazione del cliente e quindi il successo del progetto

3. Specifica dei requisiti — i dati raccolti durante la fase precedente vanno documentati all'interno dell'*Analisi dei requisiti* e devono essere ordinati e classificati.

4. Validazione dei requisiti — viene assicurato che i requisiti individuati rappresentino il quanto richiesto dal cliente e quindi bisogna che siano chiari e ben strutturati

Non bisogna limitarsi solo ai requisiti di una azienda perché il richiedente non sa tutte le potenzialità che possiamo offrire, lui ha una visione limitata. I requisiti vanno tracciati in tabelle che li identificano e li classificano univocamente.

6. Progettazione software

Dopo aver individuato il problema tramite ingegneria dei requisiti, bisogna trovare la soluzione per risolvere il problema nel modo più adatto e dunque si passa alla fase di progettazione. La progettazione ricerca una soluzione soddisfacente per gli stakeholder producendo l'**architettura** ed i suoi **modelli** senza usare codice.

- Architettura — è un *sistema* formato da *componenti* che raggruppano delle *unità* che a loro volta raggruppano dei *moduli*. Il modulo è quindi l'unità più piccola e rappresenta il carico di lavoro da assegnare ad una persona

L'acronimo di questa suddivisione è **SCUM**, le iniziali di ogni parte, utile anche farle ricordare. Per perseguire coerenza e consistenza nell'architettura è consigliato l'uso di *stili architetturali*. La progettazione passa attraverso due attività:

- Progettazione architetturale — descrive come il software viene organizzato in componenti
- Progettazione di dettaglio — descrive il comportamento di tali componenti

SEMAT prevede quattro stadi di progresso con cui poter valutare il grado con cui l'architettura è stata definita ed implementata:

1. Architecture selected — selezione di tecnologie necessarie e decisioni architetturali
2. Demonstrable — architettura approvata e stabilita la sua configurazione
3. Usable — architettura utilizzabile che ha superato il processo di sviluppo
4. Ready — stadio finale raggiungibile dopo aver creato la documentazione e dopo aver ricevuto l'accettazione da parte degli stakeholders

Le qualità che dovrebbe avere una buona architettura dovrebbero essere misurabili di così si possono evidenziare i cambiamenti nel tempo (facendo appunto delle misurazioni). Ecco alcune qualità misurabili:

- Basso accoppiamento — parti distinte dipendono il meno possibile le une dalle altre
- Affidabilità — alto grado di probabilità che svolga bene il suo compito
- Flessibilità — che permetta facile manutenzione adattiva ed evolutiva
- Modularità — scomposizione in componenti chiare distinte fra loro che non hanno compiti sovrapposti
- Semplicità — caratteristica riferita alla complessità dell'architettura

L'accoppiamento va minimizzato e per far ciò si utilizzano le interfacce. Il grado di accoppiamento è misurabile: per ogni componente il numero di archi entranti è il **fan-in** (indice di utilità, va massimizzato) mentre il numero di archi uscenti è il **fan-out** (indice di dipendenza, va minimizzato).

Gli stili di progettazione sono sostanzialmente tre:

1. Top-down — decomposizione di problemi partendo dalle funzionalità più generiche e scendendo poi nel dettaglio (facendo rifiniture)
2. Bottom-up — si identificano i dettagli e poi si connettono fra loro per formare componenti più grandi (fino ad arrivare all'intero sistema)

3. Meet-in-the-middle — approccio intermedio agli altri due che permette maggiore flessibilità

A supporto della progettazione architetturale ci sono i *framework*, che possono indurre ad usare *design pattern*, che possono influenzare delle scelte di progettazione.

- Framework — insieme di componenti software prefabbricate che danno una base facilmente riutilizzabile per un grande dominio di casistiche. Forniscono un approccio *bottom-up* perché sono codice già fatto ma anche *top-down* poiché a volte obbligano la scelta di stili architetturali

- Design pattern — soluzione progettuale generale ad un problema ricorrente che lascia comunque un certo grado di libertà di progettazione

Nella **progettazione architetturale** ritroviamo gli stili architetturali (ad esempio *client-server*, *peer-to-peer*, *MVC* ...) mentre nella **progettazione di dettaglio** ritroviamo i design pattern (divisi in creazionali, strutturali e comportamentali).

7. Documentazione

Documentare è essenziale per replicare il prodotto senza dover coinvolgere il creatore. Serve anche al gruppo per gestire lo svolgimento die processi produttivi e la loro qualità. Per il progetto sarà necessario realizzare i seguenti documenti:

- Studio di fattibilità — analisi sui capitolati disponibili con valutazione di: pregi, difetti e fattibilità in termini di costi e tempistiche
- Norme di progetto — linee guida per la gestione dei processi istanziati dal gruppo e devono seguire la struttura proposta dall'ISO/IEC 12207
- Piano di progetto — indicazioni su scadenze temporali che il gruppo prevede di rispettare e fornisce un preventivo dei costi da presentare la proponente. Uso dei diagrammi di Gantt
- Piano di qualifica — uno dei documenti più importanti che deve dare tutte le informazioni per il controllo di qualità di processi e prodotti basandosi sulla misurabilità
- Analisi dei requisiti — descrizione degli attori del sistema con casi d'uso
- Manuale utente — aiuta l'utente nell'uso del sistema e può essere più o meno dettagliato a seconda della tipologia di utente al quale è destinato
- Manuale sviluppatore — spiega ad eventuali sviluppatori esterni come interagire con le componenti del sistema ed estenderle o correggerle

Nell'AdR il tracciamento dei requisiti è importante perché motiva l'esistenza di un requisito, spiegandone l'origine (Capitolato o Interno). Conviene tracciarli in modo automatico per garantire la correttezza.

8. Qualità del software

La qualità va misurata in modo oggettivo ed è un indice importante sia per chi realizza il prodotto (usa la qualità per migliorare il prodotto) sia per chi usa il prodotto (usa la qualità per avere garanzia che funzioni tutto bene). Secondo l'ISO la qualità è:

> *L'insieme delle caratteristiche di un'entità che ne determinano la capacità di soddisfare le esigenze espresse ed implicite*

Secondo l'ISO il **sistema di qualità** è l'insieme di tutti quei fattori che servono a raggiungere la qualità, come ad esempio struttura organizzativa, responsabilità, risorse e procedure. Questo sistema gestisce la qualità di tre ambiti:

- Pianificazione — le attività che fissano gli obiettivi di qualità

- Controllo — le attività che si assicurano che il prodotto soddisfi i requisiti
- Miglioramento continuo — secondo lo schema PDCA (*Ciclo di Deming*)

Quello che si definisce **standard di qualità** è una raccolta di *best practices* che serve ad evitare la ripetizione di errori fatti nel passato. Il modello più rilevante è l'*ISO/IEC 25000:2005* che racchiude al suo interno altri due standard, considerati obsoleti e dunque messi assieme e migliorati.

Il modello *ISO/IEC 25000:2005*, conosciuto anche come **SQuaRE**, ha lo scopo di creare un *framework* per la valutazione della qualità del software. Le due parti che lo formano sono le seguenti:

- ISO/IEC 9126 — si tratta di un modello basato su 7 caratteristiche e 31 sotto-caratteristiche per valutare la qualità. Le descrive e spiega anche *come* misurarle (assegna una metrica ad ogni caratteristica). Questo modello propone anche tre visioni della qualità:

    1. Esterna — comportamento del software durante l'esecuzione e la qualità viene rilevata attraverso i test
    2. Interna — qualità del codice sorgente e della documentazione correlata
    3. In uso — punto di vista dell'utente finale sul software e necessita che la qualità interna ed esterna abbiano raggiunto un buon grado di qualità

- ISO/IEC 14589 — fornisce delle linee guida per associare una metrica ad una sotto-caratteristica dello standard ISO/IEC 9126. Si basa sul fatto che le proprietà del software possano essere misurabili e che ci possa essere una relazione fra quello che misuriamo e quello che ci interessa ricavare da queste misurazioni.

Una **metrica** è un modo per dare un significato a dei valori; una **misurazione quantitativa** è l'uso di una metrica per dare un valore utilizzando una particolare scala predefinita. Con le metriche posso quantificare sia un prodotto che un processo.

9. Qualità di processo

Il controllo della qualità di processo influenza direttamente la qualità del prodotto realizzato perché come detto all'inizio, un processo porta ad un *prodotto software*. Un processo è una macchina che prende in input bisogni e da in output dei prodotti, consumando risorse nel farlo.

Le norme **ISO 9000** rappresentano un insieme di standard che riguardano i sistemi di gestione della qualità; è stato definito poi un *Sistema di Gestione Qualità* (SGQ) in **ISO 9001** specifica i requisiti di ISO 9000 e serve a garantire la qualità nei vari settori ed è documentato. Alcuni strumenti per il miglioramento e la valutazione della qualità di processo sono:

- SPY — valuta in modo oggettivo la maturità dei processi di un'organizzazione. Il problema sta nel fatto che è completamente slegato rispetto ai modelli standard per la qualità dei processi.

- CMMI — il nome deriva dalle iniziali dei 4 principi che governano questo strumento:
    1. Capability: misura quanto un processo è adeguato rispetto allo scopo per cui è stato definito. Più alto è questo livello, maggiore è la garanzia che il processo venga eseguito da tutti in modo disciplinato.
    2. Maturity: misura quanto è governato un insieme di processi; è il risultato del livello di capability dei singoli processi presi in considerazione.
    3. Model: insieme di requisiti sempre più stringenti che valutano il percorso di miglioramento
    4. Integration: come sono state messe assieme e combinate le varie componenti

Da notare che *capability* riguarda il processo singolo mentre la *maturity* riguarda un gruppo di processi. CIMMI definisce cinque livelli di maturità che si appoggiano allo schema PDCA

- SPICE — scompare il concetto di *maturity* con i 5 livelli e si valuta in modo più accurato la *capability* dei processi. Fornisce un framework per definire gli obiettivi e rendere più facile il raggiungimento. Individua 9 attributi per misurare la *capability* e dunque questo strumento dà importanza al singolo processo.

10. Verifica e validazione

Verifica e validazione sono due processi strettamente correlati e spesso vengono indicati con "V&V"; hanno però due significati diversi che conviene ricordare:

- Verifica — è un processo che si applica ad ogni **fase** del ciclo di vita del software per accertarsi che le attività svolte in quel periodo di tempo non abbiano introdotto degli errori (*bug*).

- Validazione — è una conferma finale che il prodotto sia conforme alle attese del committente. La validazione è possibile solo dopo avere fatto delle verifiche.

La verifica è un processo che si può fare in due forme: **statico** (senza eseguire il software) oppure **dinamico** (eseguendo il software o una sua parte). Si fa prima l'analisi statica, sul codice sorgente, e poi quella dinamica, con i test di unità, perché quest'ultima ha bisogno di avere il software che sia già eseguibile, almeno in parte.

Analisi statica e dinamica fanno parte della **quality assurance**, cioè la qualità di garanzia. Questa attività è un controllo che viene fatto *prima* di fare per assicurare la qualità il più presto possibile. Si basa sull'ISO/IEC 9126 per le caratteristiche interne ed esterne.

Più tardi si inizia a fare la verifica, più difficile diventa poi individuare e correggere gli errori perché il prodotto cresce di complessità. Vediamo ora in dettaglio i due tipi di analisi:

- Analisi statica — separare interfaccia ed implementazione e massimizzare l'incapsulamento aiuta a restringere lo scope e quindi la lettura del codice diventa molto più facile. L'analisi statica può essere fatta da un umano leggendo il codice ed i metodi principali sono due:

    - Walkthrough — "non so cosa cerco, ma cerco ovunque" (attraversamento a pettine). Si fa una lettura critica "a largo spettro" senza assunzioni particolari e si va alla ricerca di errori. Dopo la pianificazione e la lettura, va discusso quanto individuato per correggere i difetti.

    - Inspection — "cerco qualcosa di specifico". Si fa una lettura mirata da parte dei verificatori alla ricerca di errori noti. Dopo della pianificazione, si crea una lista di controllo che va costantemente aggiornata.

    L'*inspection* è la tecnica più rapida da eseguire ma non è esaustiva perché si basa su una lista che potrebbe non essere completa. Il *walkthrough* richiede più attenzione perché è una ricerca a campo aperto senza presupposti e dunque richiede "occhio". L'organizzazione del codice (design pattern, architetture...) facilita o meno l'analisi statica.

    Importante è il tracciamento che deve essere automatizzato e viene creato durante lo sviluppo (ramo discendente modello a V) e serve in ogni passaggio della verifica (ramo ascendente modello a V)

- Analisi dinamica — Si tratta dell'esecuzione di test (prove) che verificano il comportamento dinamico del programma in certe situazioni. L'analisi dinamica rileva la presenza di malfunzionamenti ma **NON** ne garantisce l'assenza. Cerchiamo dei malfunzionamenti (**failures**), che danno origine ad errori (**errors**), derivati da un guasto (**fault**) meccanico o algoritmico.

  I criteri da seguire per definire i test vanno inseriti nel **Piano di Qualità**. Un test ha lo scopo d far fallire il programma e se ci riesce, quel test va aggiunto alla suite ufficiale di test del progetto in modo permanente.

  L'analisi dinamica aiuta a stimare il grado di evoluzione del prodotto analizzando quanto migliora dopo delle prove. A supporto dell'analisi dinamica ci sono:

  - Driver — un *pilota* che guida l'esecuzione del test e si occupa di chiamare i test di unità
  - Stub — un *calco* che sostituisce codice non ancora scritto e simula una parte del sistema
  - Logger — scrive l'esito di una prova su di un file

  Vediamo ora i diversi tipi di test che ci possono essere:

  - Test di unità — testano parti di software e vanno definiti durante la progettazione di dettaglio; circa 2/3 dei difetti trovati dall'analisi dinamica derivano da questo tipo di test.
  - Test di integrazione — verificano la corretta interazione fra le componenti del sistema e si basano sui componenti specificati dall'architettura
  - Test di sistema — verificano la copertura dei requisiti e vengono definiti durante il periodo di analisi dei requisiti.
  - Test di regressione — ripetizione di test di integrazione ed eventualmente di sistema per accertare che le modifiche (correzioni o estensioni) non abbiano introdotto errori.

Un concetto fondamentale nei test di unità è quello di **copertura** (*coverage*). Con questo termine si intende la percentuale di codice che un test è in grado di eseguire, cioè quanto codice è stato "coperto" dal test. Una copertura del 100% indica che l'esecuzione di un test ha coperto tutti i casi possibili di un pezzo di codice scelto in esame. Alcuni criteri di copertura sono:

- Function coverage — quante funzioni sono state eseguite
- Statement coverage — quante istruzioni sono state eseguite
- Branch coverage — quanti rami del programma sono stati eseguiti
- Condition coverage — quanti valori di espressioni logiche sono stati testati

11. Metodi e obiettivi di quantificazione

La **misurazione** è il processo che assegna numero o simboli a entità del mondo reale e produce delle **misure**; un insieme di regole che interpretano delle misure si chiama **metrica**. Si utilizzano le metriche per valutare la qualità dei prodotti e fare delle stime o comunque delle considerazioni.

Ecco alcuni esempi di metriche **architetturali** (**progettuali**):

- Instability — (vedi capitoli sopra) è il rapporto fra il *fan-out* (archi uscenti) ed il *fan-in* (archi entranti) e indica la possibilità di poter fare modifiche ad una componente senza influenzarne altre
- Grado di coesione — quanto sono coese le funzionalità di una classe o di un modulo; alta coesione significa avere un *fan-in* elevato (e va bene, l'approccio OOP vuole un **alto** grado di coesione)

- Grado di accoppiamento — quanto un modulo è dipendente da altre componenti del sistema; alto accoppiamento significa avere un *fan-out* elevato (e va male, l'approccio OOP vuole un **basso** grado di accoppiamento)

Ecco alcuni esempi di metriche **software**:

- Complessità ciclomatica — indica il numero di cammini indipendenti che l'esecuzione di un metodo può intraprendere. Un valore alto indica che un metodo è troppo complesso, scarsamente modulare e manutenibile.
- Nested block depth — misura il massimo numero di livelli di annidamento delle strutture di controllo dei metodi

Ecco alcuni esempi di metriche per la **gestione di progetto**:

- Budget variance — misura la differenza tra i costi preventivati e quelli sostenuti fino ad ora
- Schedule variance — misura la differenza tra il tempo attualmente speso rispetto a quello preventivato

by Roberto