Glossario di Ingegneria del Software

a.a. 2018-2019

Indice

A 4

B 9

C 11

D 17

E 19

F 20

G 21

I 23

J 24

L 25

M 26

N 38

O 39

P 40

Q 51

R 52

S 56

T 62

U 68

V 69

W 71

Z 72

Indice analitico

Amministratore, 4
Analisi dei requisiti, 4
Analista, 6
Approccio, 7
Architecture Selected, 7
Architettura, 7
Audit Process, 8

Backlog, 9
Baseline, 9
Best practice, 9
Body of knowledge, 9
Brainstorming, 10
Branch Coverage, 10

Cammino critico, 11
Capability, 11
Capitolati d'appalto, 11
Ciclo di Deming, 11
Ciclo di vita, 11
Classificazione dei requisiti, 12
CMMI, 12
CoCoMo, 13
Coeso, 13
Collaudo, 13
Committente, 13
Componente, 13
Computational thinking, 14
Configurazione, 14
Considerazioni pragramtiche, 14
Consuntivo, 14
Controllo, 14
Controllo di configurazione, 14
Controllo di Qualità, 15
Controllo di versione, 14
Correttezza, 15
Coverage, 15
Criteri di programmazione, 15
Customer, 16

Demonstrable, 17
Diagramma di Gantt, 17
Diagramma di PERT, 17
Diminishing returns, 18
Divide et Impera, 18
Documenti, 18

Driver, 18

Economicità, 19
Efficacia, 19
Efficienza, 19
Endeavor, 19
Engineering, 19
Error, 19

Failure, 20
Fase, 20
Fault, 20
Framework, 20

Gestione dei cambiamenti, 21
Gestione dei requisiti, 21
Gestione dei rischi, 21
Gestione della configurazione, 21
Gestione di progetto, 22
Gestione di qualità, 22

Implementation, 23
Incapsulamento, 23
Incremento, 23
Inspection, 23
Integrazione continua, 23
Iterazione, 23

Joint Review Process, 24

Lazy, 25
Lead time, 25
Logger, 25

Manuale della qualità, 26
Manuali, 26
Manutenzione, 26
Maturità, 26
Metrica, 27
Milestone, 28
Mistake, 28
Misura, 28
Mock, 29
Modelli di sviluppo, 29
Modello a evoluzioni successive, 32
Modello a spirale, 37
Modello agile, 37
Modello incrementale, 30

Modello iterativo, 31
Modello per componenti, 32
Modello SCRUM, 37
Modello sequenziale, 30
Moduli, 37

Norme di Progetto, 38

Oppurtunity, 39
Oracolo, 39

Pianificazione, 40
Piano di progetto, 40
Piano di Qualifica, 41
Portabilità, 41
PPP, 41
Preventivo, 41
Principio del miglioramento continuo, 41
Problem solution, 41
Processo, 42
Prodotto software, 43
Product Baseline, 44
Produttività, 44
Progettazione, 44
Progettista, 46
Progetto, 46
Programmatore, 48
Programmi verificabili, 48
Proof of Concept, 49
Prototipo, 50
Prova, 50

Qualifica, 51
Qualità, 51

Ready, 52
Realizzazione, 52
Requirements, 52
Responsabile, 53
Revisione dei Requisiti, 53
Revisione di Accettazione, 54
Revisione di Progettazione, 54
Revisione di Qualifica, 54
Riuso, 55
Ruoli, 55

Scalabilità, 56
SEMAT, 56
Sistema di qualità, 56
Sistema software, 56
Slack, 56
Software deterministico, 56
Software Engineering, 56
Solution, 56
Stakeholder, 57
Standard di processo, 57
Standard IEEE 830-1998, 57
Standard ISO 12207, 57
Standard ISO 14598, 59
Standard ISO 15504, 59
Standard ISO 15939, 59
Standard ISO 25000, 59
Standard ISO 9000, 59
Standard ISO 9001, 60
Standard ISO 9126, 60
Statement Coverage, 60
Stato, 60
Stub, 60
Studio di fattibilità, 61

Task, 62
Technology Baseline, 62
Tempo/persona, 62
Test, 62
Test case, 65
Test di integrazione, 65
Test di regressione, 66
Test di sistema, 66
Test di unità, 66
Test suite, 66
Top-down, 66
Tracciamento, 66

Unità, 68
Usable, 68

Validare, 69
Verificare, 69
Verificatore, 70
Versione, 70

Walkthrough, 71
WBS, 71

Zero-latency, 72
Zero-laxity, 72

A

AMMINISTRATORE

È uno dei ruoli in un progetto. In breve si occupa di controllare l'ambiente di lavoro. Nello specifico di:

- Amministrare le infrastrutture di supporto
- Risolvere problemi legati alla gestione dei processi
- Gestire la documentazione di progetto
- Controllare versioni e configurazioni

ANALISI DEI REQUISITI

L'attività di analisi dei requisiti è lo step dopo lo studio di fattibilità (stesura documento Studio di Fattibilità) e tratta sostanzialmente di capire appieno il problema. Riguarda la qualifica e se ne occupa l'Analista che deve cercare di entrare nell'ottica dell'utente.
Lo *svolgimento* dell'analisi prevede:

- Lo studio dei bisogni e delle fonti del dominio applicativo
- Una prima classificazione dei requisiti
- Una modellazione concettuale del sistema
- L'assegnazione dei requisiti alle varie parti del sistema
- La negoziazione con il committente

Dopodiché avviene la redazione del Piano di qualifica per metodi, tecniche, strumenti, tempi ecc.
Le *attività* di analisi sono:

- Studiare e definire il problema da risolvere, identificando il prodotto da commissionare, capendo cosa deve essere realizzato e definendo gli accordi contrattuali (compito del cliente)
- Verificare il costo e la qualità in base ai requisiti derivanti dal cliente
- Studio dei bisogni e delle fonti, identificando, specificando e classificando i requisiti (lato fornitore)
- Modellazione concettuale del sistema con partizionamento in componenti (ambiti) a scopo di allocazione dei requisiti, per esempio con diagrammi d'uso (lato fornitore)
- Ripartizione dei requisiti a parti del sistema (lato fornitore)
- Accertare la soddisfacibilità dei requisiti rispetto ai vincoli di processo
- Assicurare, tramite tracciamento, che i requisiti concordati siano tutti e soli quelli necessari (tutti i requisiti in AdR soddisfano un particolare bisogno) e sufficienti (tutti i bisogni rilevati nelle fonti sono requisiti in AdR)

- Determinare con il cliente l'utilità strategica dei requisiti concordati
- Adozione di norme redazionali (stesura documento Norme di Progetto), perchè aiuta a evitare espressioni ambigue, e glossario, perchè aiuta a garantire terminologia consistente
- Uso di metodi (semi-)formali, come diagrammi e formule, perchè aiuta a ridurre gli errori di interpretazione

I *processi di supporto* implicati da esse sono:

- Documentazione, al fine di raccogliere i risultati dello studio di fattibilità e specificare i requisiti
- Gestione e manutenzione dei prodotti, comprendente il tracciamento dei requisiti, impostazione e gestione della configurazione e dei cambiamenti

All'interno del nostro progetto possiamo quindi dire di avere diversi prodotti documentali: per *definire* i bisogni (utente e SW) abbiamo il capitolato d'appalto, per *specificare* abbiamo appunto l'Analisi dei Requisiti (che è un documento contrattuale) e lo Studio di Fattibilità (che è un documento interno del fornitore).
Per la *ripartizione* dei requisiti invece la questione è molto delicata perchè da qui inizia la Progettazione. Il confine tra Analisi e Progettazione è molto sottile: per esempio, l'Analista riesce già a vedere dei sotto-problemi che poi però sono compito del Progettista.

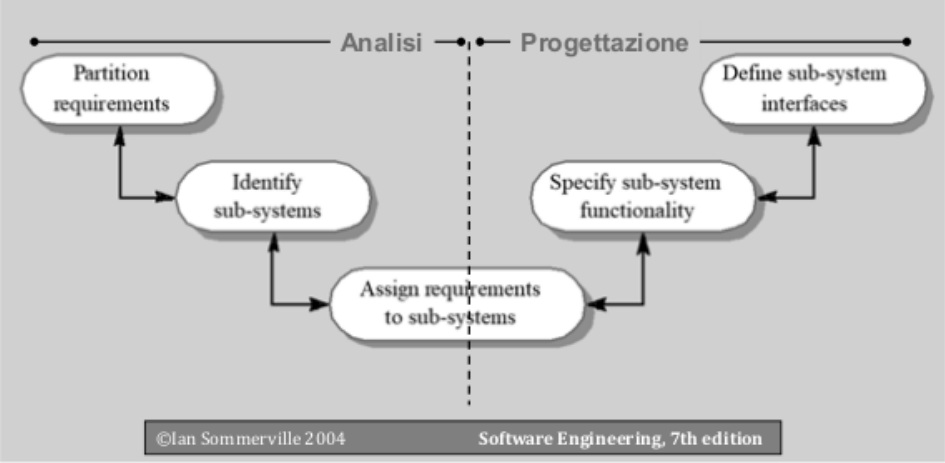


Figura 1: Confine tra Analisi e Progettazione.

Ci sono diversi approcci su come procedere per l'Analisi dei Requisiti:

- **Top-down**
- **Bottom-up**
- **Agile**: generalmente parto da un'idea di una cosa che già esiste e penso alle funzionalità da poterci aggiungere. È una via di mezzo tra le altre due.

Le *tecniche* di analisi comprendono:

- L'interazione con il cliente (il cui esito viene documentato in un verbale), tramite interviste o discussione di scenari
- Discussioni creative e collaborative, ovvero brainstorming
- Prototipazione, interna (per il fornitore) o esterna (per la discussione con il cliente)
- La comprensione del dominio, che prevede:
  - Una serie di domande base come *A quali bisogni risponde il prodotto atteso?* e *Quali problematiche d'uso esso comporta*
  - L'acquisizione delle conoscenze tramite documentazione preesistente e interviste ai potenziali utenti
  - Consolidamento del glossario che raccoglie e definisce i termini chiave del dominio per avere un'interazione ordinata con il committente

Capita a volte che il progetto venga abbandonato e le principali cause sono:

- Requisiti incompleti
- Insufficiente coinvolgimento del cliente e/o dell'utente (non sono necessariamente la stessa entità)
- Scarsità di risorse
- Attese irrealistiche
- Insufficiente competenza tecnologica e/o metodologica del fornitore

Mentre gli stati di progresso secondo SEMAT sono:

- *Conceived*: il committente è identificato e gli stakeholders vedono sufficienti opportunità per il progetto
- *Bounded*: i macro bisogni sono chiari e i meccanismi di gestione dei requisiti (cambiamento e configurazione) sono fissati
- *Coherent*: i requisiti sono classificati e quelli essenziali (obbligatori) sono chiari e ben definiti
- *Acceptable*: i requisiti fissati definiscono un sistema soddisfacente per gli stakeholder
- *Addressed*: il prodotto è pronto al rilascio e all'uso
- *Fulfilled*: il prodotto merita la piena approvazione degli stakeholder tanto soddisfa i requisiti

Una struttura per il documento di Analisi dei Requisiti è definita dallo standard IEEE 830.

ANALISTA

È uno dei ruoli in un progetto. Generalmente sono pochi ma hanno molta influenza sul successo del progetto. Conosce il dominio del problema e ha esperienza professionale, ma raramente segue il progetto fino alla consegna.

APPROCCIO

Avvicinarsi/predisporsi. Può essere:

Sistematico

ovvero lavorando in maniera metodica, cioè ho un metodo da seguire che mi precede, e rigorosa usando ed evolvendo la best practice. In questo è importante il tempo.

Disciplinato

ovvero seguendo regole fissate. Per essere disciplinato ho bisogno di due documenti: Norme di Progetto per fissare il metodo, Piano di Progetto per sapere quando fare.

Quantificabile

ovvero che permette di misurare efficienza ed efficacia. Il nostro fare deve produrre cose buone **oggettivamente**. Ci interessa misurare la qualità, per vedere il raggiungimento degli obiettivi che mi sono data.

ARCHITECTURE SELECTED

L'architettura viene capita e selezionata, oltre alla selezione delle tecnologie necessarie.
È il primo degli stati di progresso del SEMAT per la progettazione.

ARCHITETTURA

Si intende architettura logica: è di alto livello (quindi non implementato, concettuale) e consiste nel dividere in parti per massimizzare il parallelismo. L'obiettivo è dividere fino a che non se ne trae più vantaggio, ovvero fino a che il costo della divisione diventa più un onere che un beneficio. Riporta UNA soluzione che soddisfa il cliente. Caratteristiche dell'architettura:

- Decomposta (suggerisce l'idea di top-down) in componenti
- Ha un'organizzazione: le componenti stanno insieme secondo regole date e ognuna ha un ruolo preciso e collabora
- Definisce le interfacce: qual è il modo in cui le componenti collaborano. Essa è associata a protocollo (= accordo) perché un'interfaccia si appoggia ad un protocollo
- Paradigmi (= come si fa) di composizione: componenti messe insieme secondo regole, limiti, vincoli. Definisce come vengono organizzate

Le architetture hanno scelte di paradigmi ed esistono più stili architetturali che determinano l'organizzazione dell'informazione di stato e l'interazione tra le parti.
Le qualità di una buona architettura sono:

- **Modularità**: (legata a Incapsulamento e Disponibilità) ha l'obiettivo di minimizzare la dipendenza tra parti. Ha due opzioni:
  1. Suddivide, come la pipeline è divisa in stadi

2. In modo resiliente, ovvero facendo *information hiding* altrimenti si espone ciò che è "implementation detail" (deve essere ben nascosto perchè altrimenti confonde e dà disagio)

- **Sufficienza**: deve soddisfare tutti i requisiti, coprire il bisogno
- **Comprensibilità**: deve essere capita dagli stakeholder, anche perché ci sono diversi stili
- **Robustezza**: sta in piedi anche se cambio un modulo
- **Flessibilità**: non deve collassare, deve essere in grado di evolvere (attuando modifiche a costo contenuto)
- **Riusabilità**: fare nell'intento che possa essere buono anche per altri in futuro
- **Efficienza**: senza eccesso di risorse
- **Affidabilità**: quando c'è bisogno di utilizzarla, fa le cose che deve
- **Disponibilità**: sta in piedi senza grande bisogno di manutenzione (se una parte è sotto manutenzione, non deve essere interrotto tutto il sistema)
- **Sicurezza rispetto a malfunzionamenti**: *"safety"*, non ho malfunzionamenti che fanno danno
- **Sicurezza rispetto a intrusioni**: *"security"*, invulnerabile rispetto alle intrusioni. Mitigare il rischio e massimizzare i vantaggi
- **Semplicità**: le parti contengono solo il necessario, niente di superfluo
- **Incapsulamento**: (*Information hiding*) l'interno delle componenti non è visibile dall'esterno, quindi i clienti conoscono solo l'interfaccia. Aumenta la manutenibilità e la possibilità di riuso
- **Coesione**: le parti che stanno insieme hanno gli stessi obiettivi e ognuna ha un ruolo
- **Basso accoppiamento**: l'accoppiamento è la nemesi della coesione, perchè quando viene mossa una parte ne viene conseguentemente mossa anche un'altra. Vogliamo quindi distinte parti che dipendono poco o niente le une dalle altre. È un probelma però quando dall'esterno si fanno assunzioni su come certe cose stiano all'interno di altre e quando si condividono frammenti delle stesse risorse: bisogna cercare di massimizzare l'indice di utilità e minimizzare l'indice di dipendenza

Tutte le qualità delle caratteristiche attese vanno perseguite e tutto va sottoposto a verifica.

AUDIT PROCESS

Imporre un andamento e assicurarsi che l'attività che si sta facendo si svolga nel migliore dei modi possibili. Permette di migliorare il way of working. Si tratta di una revisione **esterna** (in particolare RR e RA).

B

BACKLOG

È chiamato anche "to do". Sono una lista di cose da fare per soddisfare la *user story*, ovvero l'informale descrizione delle caratteristiche che il prodotto software deve avere.

BASELINE

Letteralmente "campo base" o "punto d'appoggio", per evitare situazioni di rischio. È un risultato concreto che risponde in maniera affermativa alla domanda:"Si può fare?". È una risposta buona e possibile in quel dato momento.

- **Cos'è**
  Un progetto software, se si basa su una strategia, avrà una sequenza di obiettivi. Le parti di cui è fatta una baseline (le quali hanno un numero di versioni "as many as needed"), esistono perché assolvono un obiettivo. Una baseline rappresenta quindi un punto di avanzamento consolidato in un dato istante del progetto. Ogni punto di avanzamento viene fissato precedentemente in modo strategico dalla best practice, ma il numero di baseline non è deciso a priori: solo gli obiettivi sono decisi a priori. È un insieme di *configuration item*.

- **A cosa serve**
  Dare una base da cui partire per l'avanzamento del progetto.

- **Come si mantiene**
  Bisogna decidere come le parti concorrono a formare la baseline tramite versionamento (perchè per esempio se arrivo ad una versione 2.9.0 e mi rendo conto che sto sbagliando tutto, posso tornare facilmente alla versione 2.0.0).

- **Come si costruisce**
  Tramite configurazione.

Concetto legato a milestone.

BEST PRACTICE

Modo di fare che deve garantire i migliori risultati in specifiche e note circostanze.

BODY OF KNOWLEDGE

"Corpo"/Insieme di conoscenze che ci ha emancipato.

BOTTOM-UP

Concepisco il sistema basandomi dalle ipotetiche parti che possono comporlo. Questo approccio è tipico della *Programmazione ad Oggetti* (esempio di: costruisco il frigo sapendo che esso è un aggregato di reparti).

BRAINSTORMING

Pensiero intenso collettivo che fa nascere idee, in cui ognuno parla a turno e non c'è sopraffazione. Durante la discussione, una sola persona scrive.
Questo è uno strumento utile per unire debolezze in un'unica forza maggiore.

BRANCH COVARAGE

Si occupa di coprire i rami di decisione. Si ha copertura al 100% quando ciascun ramo del flusso di controllo dell'unità viene attraversato almeno uno volta, ognuno con esito corretto.
Il valore di copertura è determinato dalla complessità di espressioni di decisione (espressioni composte da condizioni contenenti valori booleani).

C

CAMMINO CRITICO

Sequenza di attività ordinata con prodotto importante e dipendenze temporali strette. Miro al cammino di peso massimo.

CAPABILITY

Traduzione: possibilità/capacità.
Misura l'adeguatezza di un processo per gli scopi a esso assegnati. È una caratteristica propria del processo e determina il risultato (in termini di efficienza ed efficacia) raggiungibile per quel processo. Conviene che il livello di capability sia alto, ovvero seguito da tutti in modo disciplinato, sistematico e quantificabile.

CAPITOLATO D'APPALTO

Documento tecnico che descrive in maniera dettagliata tutti i bisogni. In esso vengono spiegate le cose che chi commissiona vuole, nel gergo di una persona normale. Sono interamente responsabilità del cliente e da esso ne conseguono:

- **Requisiti utente** che sono vincoli contrattuali che specificano il *cosa*
- **Requisiti software** che specificano il *come*

CICLO DI DEMING

Il ciclo di Deming (o ciclo di PDCA) è un metodo di gestione iterativo per il controllo e il miglioramento continuo dei processi e dei prodotti, suddiviso in 4 fasi:

- **Plan**: definisce attività, scadenze, ecc. necessari a raggiungere specifici obiettivi di miglioramento
- **Do**: esegue le attività di *Plan*
- **Check**: verifica l'esito delle azione di miglioramento rispetto alle attese
- **Act**: consolida il tutto e cerca dei modi per il miglioramento successivo

CICLO DI VITA

Si riferisce alla completa durata del prodotto, dal concepimento al ritiro (= fine) che deve essere garantita. È da vedersi come una macchina a stati in cui ho una certa sequenza di passaggi da seguire. La transizione tra stati avviene tramite l'esecuzione di attività di processi di ciclo di vita. Lo stazionamento in uno stato o transizione viene detta fase. In seguito al ritiro, il prodotto può essere "rimesso in vita", per questo viene chiamato *ciclo*.

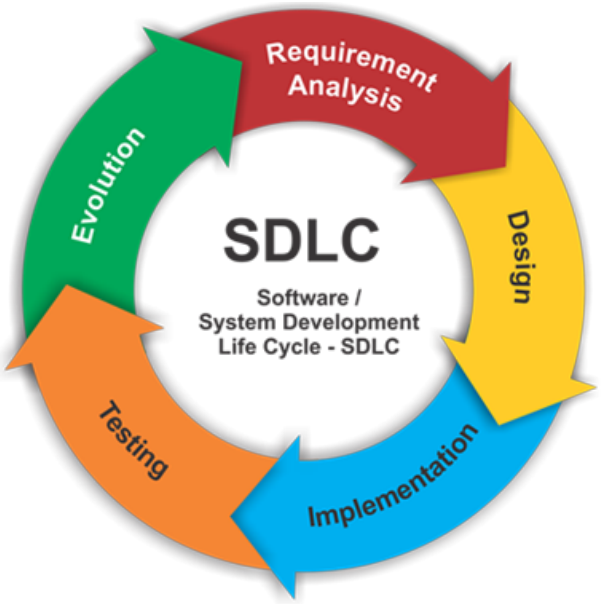


Figura 2: I 5 stati del ciclo di vita di un prodotto.

Associare un sistema di qualità al modello adottato, aiuta a perseguire conformità nel progetto e maturità nei processi. Conoscere il ciclo di vita previsto di un prodotto, aiuta a valutarne preventivamente i tempi, i costi e i rischi.
Esistono:

- Processi di ciclo di vita, che specificano le *attività* da svolgere per abilitare le transizioni di stato nel ciclo di vita
- Modelli di ciclo di vita, che descrivono questi processi e come concorrono ad abilitare le specifiche transizioni

Per organizzare le attività dei processi implicati, è necessario identificare le dipendenze tra gli ingressi dell'uno e le uscite dell'altro, in modo da fissare quindi l'ordinamento nel tempo e i criteri di attivazione e completamento.

CLASSIFICAZIONE DEI REQUISITI

Guardo in generale i requisiti e li classifico in base a:

- **Cosa devo fare** con il prodotto, quindi secondo gli attributi del prodotto (ho dei requisiti funzionali)
- **Come devo farlo**, quindi secondo i processi (ho dei requisiti di vincolo)

CMMI

CMM (*Capability Maturity Model*) evoluto poi in CMMI (*Capability Maturity Model Integration*) è un *modello di valutazione* (standard) uniforme dei fornitori.
Model è l'insieme di criteri per valutare il grado di qualità (in scala assoluta) dei processi dell'azienda, mentre *Integration* è l'architettura di integrazione delle diverse discipline (system, HW, SW) e tipologie di attività delle aziende.

CoCoMo

Constructive Cost Model è un modello algoritmico che stima le risorse necessarie esprimendone la misura in mesi/persona (`MP`). 1MP sono 152 ore.
Questo modello è da utilizzare per la gestione di progetto.

COESO

Ciò che è coeso indica avere delle attività messe insieme a un dato scopo, che devono esserci e se non ci fossero se ne sentirebbe la mancanza.
Per quel che riguarda le componenti di un'architettura possiamo dire che funzionalità "vicine" devono stare nella stessa componente e dato che la modularità spinge a decomporre il grande in piccolo, la ricerca di coesione fornisce un criterio di decomposizione.

La coesione *va massimizzata* per ottenere maggiore manutenibilità e riusabilità, minor legame fra componenti e maggiore comprensibilità dell'architettura del sistema. La coesione inoltre si può misurare.
Ci sono diversi tipi di buona coesione (la migliore è sempre quella che persegue *information hiding*):

- **Funzionale**: quando le parti concorrono allo stesso specifico compito.
- **Sequenziale**: quando alcune azioni sono più "vicine" ad altre per ordine di esecuzione ed è conveniente tenerle insieme.
- **Informativa**: quando le parti agiscono sulla stessa unità d'informazione (la best practice).

COLLAUDO

È l'atto formale di verifica di efficienza o di validità. Ad esso segue il rilascio del prodotto e la fine della commessa.

COMMITTENTE

Persona che ha il compito di identificare il prodotto da commissionare.

COMPONENTE

Radice della composizione, ovvero non soltanto è una parte, ma è fatto per essere messo insieme ad altre parti.

COMPORTAMENTO PREDICIBILE

Ci interessa garantire comportamento predicibile del software, ovvero che "lo posso dire prima", perché non ci sia ambiguità. Gli *elementi di vulnerabilità* per garantire comportamento predicibile del SW sono:

- **Effetti laterali** (side-effect): la causa sono variabili condivise. La soluzione è l'incapsulamento.
- **Ordine**: di due importanti momenti:

  - **Elaborazione**, ovvero preparare le risorse logiche di cui il programma avrà bisogno al principio, ad esempio avere un po' di RAM, quindi preparare l'ambiente di esecuzione
  - **Inizializzazione**, ovvero dichiarare le variabili prima di fare *begin*

- **Passaggio di parametro**: per valore e per riferimento (ricordo che in Java le variabili vengono passate ad una funzione facendone una copia e ricordare esempio dello swap che non fa effettivamente lo scambio).

COMPUTATIONAL THINKING

L'insieme dei processi mentali coinvolti nella formulazione di un problema e della sua soluzione, in modo tale che un umano o una macchina possa effettivamente eseguirlo. La cosa giusta da fare è aumentare abilità e sfide proporzionalmente.

CONFIGURAZIONE

Indica le parti di un prodotto software e come esse vengono messe insieme.

CONSIDERAZIONI PRAGMATICHE

Termine legato a programmi verificabili.
L'efficacia dei metodi di verifica è funzione della qualità di strutturazione del codice. Ad esempio una procedura con un solo punto di ingresso e un solo punto di uscita è più facilmente analizzabile per il suo effetto sullo stato.
La verifica di un programma relaziona frammenti di codice con frammenti di specifica: la verificabilità è funzione inversa dell'ampiezza del contesto. Conviene quindi confinare gli ambiti e la visibilità.

CONSUNTIVO

Rendiconto dei risultati di un dato periodo di attività di un ente o di un'impresa, che si stima verso la fine. Esistono:

- **Consuntivo di periodo**: ogni azione in questo periodo chiede nuova pianificazione sul rimanente (il prodotto è la pianificazione del residuo)
- **Preventivo a finire**: conseguente al consuntivo di periodo

CONTROLLO

Di Versione

Riguarda la gestione della storia di un prodotto.

Di Configurazione

Riguarda la frammentazione del codice in parti (ad esempio perchè non vogliamo 2000 righe di codice in un file).

CONTROLLO DI QUALITÀ

Controllo di Qualità (o *Quality Assurance*) sono le attività del sistema qualità pianificate e attuate per assicurare che il prodotto soddisfi le attese.
Si può fare in due modi:

- **Assaggiatore**: ogni prodotto realizzato lo faccio valutare (non è la miglior soluzione perchè non vado a monte del problema, potrei sprecare risorse)
- **Quality assurance**: perseguendo attivamente altrimenti è uno sforzo collettivo. Bisogna avere un way of working che mostri fiducia e controllo a monte. Ma tutto questo deve avvenire in modo non invasivo. Conviene seguire uno standard.

CORRETTEZZA

Per costruzione

"Correctness by construction" vuol dire avere strumenti oggettivi misurabili che ci dicono se stiamo andando bene. Con regole di questo tipo, sono confidente di quello che ho fatto.

Per correzione

"Correctness by correction" vuol dire provarci, cercando di sistemare pezzo per pezzo. Non sappiamo quindi effettivamente se funziona.

COVERAGE

Il *fattore di copertura* è quanto la prova fa esercitare il prodotto rispetto alla percentuale di:

- Funzionalità esercitate come viste dall'esterno: **copertura funzionale**
- Logica interna del codice esercitata: **copertura strutturale** (branch, condition)

Attenzione: una copertura al 100% non prova assenza di difetti e in ogni caso un 100% può essere irraggiungibile.

CRITERI DI PROGRAMMAZIONE

Criteri di programmazione: i *criteri* (ovvero norme fondanti) servono per riconoscere i principi che ho usato per fare incapsulamento e altre norme:

- Architettura (*design*) del codice
- Separazione interfaccia e implementazione: ciò rende bassissimo l'accoppiamento. L'idea è che il client non deve conoscere l'implementazione, ma solo l'interfaccia (che non deve cambiare)
- Massimizzazione di *information hiding*
- Uso dei tipi (se ci sono): devo fare operazioni il cui esito sia verificabile, sapendo i tipi

CUSTOMER

Primo elemento di un progetto secondo SEMAT. Comprende opportunity e stakeholders.

D

DEMONSTRABLE

Ha a che vedere con la Technology Baseline perchè tratta di essere in grado di convincere che la scelta da noi fatta è buona. In questa circostanza vengono prese le decisioni sulle principali interfacce e configurazioni.
È uno degli stati di progresso del SEMAT per la progettazione.

DIAGRAMMA DI GANTT

Il diagramma mostra la dislocazione temporale delle attività per rappresentarne la durata, la sequenzialità, il parallelismo, e confrontarne le stime con i progressi.
È uno strumento per la pianificazione.

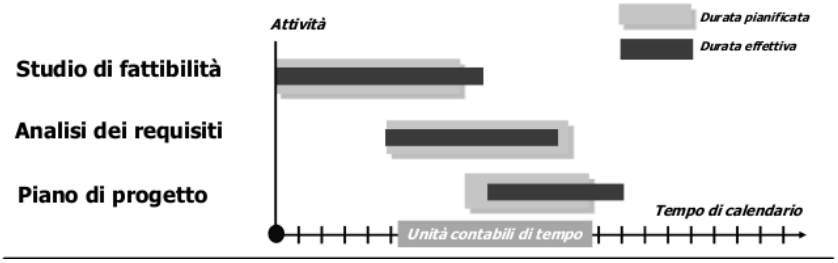


Figura 3: Esempio di diagramma di Gantt.

DIAGRAMMA DI PERT

Questo tipo di diagramma (*Programme Evaluation and Review Technique*) mostra le dipendenze temporali tra le varie attività al fine di ragionare sulle scadenze di progetto.
È uno strumento per la pianificazione.

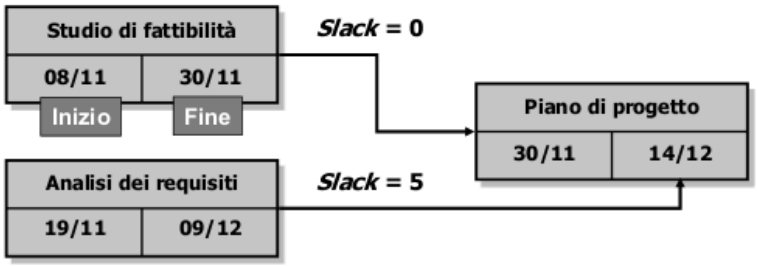


Figura 4: Esempio di diagramma di PERT.

Si parla qui di *slack time*: periodo di tempo durante il quale un'attività può essere ritardata senza ritardare l'intero progetto di cui fa parte. Si calcola facendo la differenza tra l'ultima data disponibile per compiere l'attività e la prima data disponibile perché venga compiuta. Ha a che vedere con il cammino critico.

DIMINISHING RETURNS

Ritorno in diminuzione è un certo punto in cui la curva dell'output decresce, ovvero proseguire costa più del beneficio che se ne trae.
È il punto in cui i test non rilevano più errori.

DIVIDE ET IMPERA

Se ho un prodotto fatto di parti riesco ad avere parallelismo nell'implementazione, in modo da andare n volte più veloce.

DOCUMENTI

I documenti adottano una strategia: come mi approccio al problema e come mitigo i rischi. La milestone è definita dalla strategia e applica il divide-et-impera perchè divide tutto in problemi più piccoli. Ricordo che la strategia va SEMPRE FATTA ALL'INDIETRO, che vuol dire non pianificare da ora in avanti, ma partire dalla scadenza prefissara e pianificare da quel giorno ad oggi.
Tra i documenti da consegnare nel corso delle revisioni troviamo:

- Piano di Progetto: quale strategia utilizzare, come nel tempo prepararci ad addomesticare i rischi, ecc
- Norme di Progetto
- Piano di Qualifica
- Studio di Fattibilità
- Analisi dei Requisiti
- Manuali tra cui:
  - Manuale utente
  - Manuale sviluppatore

DRIVER

È una componente attiva fittizia per pilotare il test e serve per eseguire su un'unità che non ha main.

E

ECONOMICITÀ

L'insieme di efficienza ed efficacia.

EFFICACIA

Capacità di produrre l'effetto e i **risultati voluti**. Misurabile grazie al Piano di Qualifica. (Il termine è in stretta correlazione a qualità e conformità).

EFFICIENZA

Capacità costante di rendimento e rispondenza per i propri fini **senza sprecare risorse**. Metrica di riferimento: produttività. Si ha nel Piano di Progetto, perchè misuro spartendo le risorse.

ENDEAVOR

Terzo elemento di un progetto secondo SEMAT.
È l'atto costruttivo di fare (= tentativo/provare) e comprende:

- **Work**: attività e compiti da fare.
- **Team**: perché collaborativo.
- **Way of working**: un team è un team soltanto se ha il suo modo di lavorare, cosa che sta alla base di tutto il nostro lavoro e regola i processi.

ENGINEERING

Applicazione pratica di principi scientifici e matematici. Ciò vuol dire che non *crea* bensì *attua* secondo la best practice. Comporta inoltre responsabilità etiche e professionali.

ERROR

Ha a che vedere con la gerarchia di problemi nei test.
Error è a monte della failure e accade perché lo stato del sistema è sbagliato. Può essere meccanico, algoritmico o concettuale.

F

FAILURE

Ha a che vedere con la gerarchia di problemi nei test.
Failure è l'effetto che vedo di un malfunzionamento (un effetto finale sbagliato, che non mi aspettavo).
In una versione gerarchica più grande, una failure potrebbe generare un altro fault che causa altri errori.

FASE

Segmento contiguo di durata temporale che ha inizio e fine. È diviso a sua volta in altri segmenti. Può essere:

- Di stato
- Di transizione (stato che mi porta da x a y)

FAULT

Ha a che vedere con la gerarchia di problemi nei test.
È la ragione dell'error, ovvero causa l'errore. Proviene da un mistake umano.

FRAMEWORK

Schimdt sostiene che il framework è: *an integrated set of software artifacts (such as classes, objects and components) that collaborate to provide a reusable architecture for a family of related applications.* I frameworks forniscono supporto per caratteristiche generiche che possono essere usate in tutte le applicazioni di tipo simile.

G

GESTIONE DELLA CONFIGURAZIONE

Processo di supporto implicato dall'Analisi dei requisiti che è un documento collaborativo in cui ognuno ha una determinata parte e non c'è confusione.

GESTIONE DEI CAMBIAMENTI

Processo di supporto implicato dall'Analisi dei requisiti perchè l'analisi dei requisiti non è un'attività libera da vincoli. Ogni cambiamento deve avere un resoconto.
Valuta la fattibilità tecnica e l'impatto sul progetto.

GESTIONE DEI REQUISITI

Attività da svolgere per l'Analisi dei requisiti. Prevede l'identificazione e la classificazione dei requisiti:

- Identificatore unico
- Numerazione sequenziale basata sulla struttura del documento
- Organizzazione secondo la coppia [`CATEGORIA`, `NUMERO`]

Avviene inoltre la gestione dei cambiamenti e la tracciabilità (*requisiti* in rapporto a *parti della specifica* in rapporto a *componenti del sistema*).

GESTIONE DEI RISCHI

Fa parte della gestione di progetto. Prevede:

- Identificazione dei rischi nel progetto, nel prodotto e nel mercato
- Analisi della probabilità di occorrenza e conseguenze possibili
- Pianificazione, ovvero come evitare i rischi e mitigarne gli effetti
- Controllo, ovvero prestare continua attenzione tramite rilevazione di indicatori e raffinamento delle strategie

Figura 5: Schema della gestione dei rischi.

GESTIONE DI PROGETTO

La gestione di progetto prevede:

- L'istanziazione di processi nel progetto secondo degli standard di processo
- La stima di costi e risorse necessarie tramite CoCoMo. I fattori di influenza per le stime sono:
  - Dimensione del progetto
  - Esperienza del dominio
  - Familiarità con le tecnologie
  - Produttività dell'ambiente di lavoro
  - Qualità attesa
- Pianificazione (partendo dall'obiettivo, quindi all'indietro, non dall'inizio) di attività con conseguente assegnamento alle varie persone
- Il controllo delle attività e la verifica dei risultati

In questo contesto ogni persona assume un certo ruolo e ad ogni ruolo viene assegnata un'attività (da notare che spesso molte risorse possono essere impegnate su più progetti).
La gestione di progetto prevede anche la Gestione di Qualità e un Piano di Progetto.
I principali *fattori di successo* di un progetto sono:

- Il coinvolgimento del cliente
- Il supporto della direzione esecutiva
- La definizione chiara dei requisiti
- Una pianificazione corretta
- Delle aspettative realistiche
- Un personale competente

Mentre i primi *fattori di fallimento* sono:

- Dei requisiti incompleti
- Il mancato coinvolgimento del cliente
- La mancanza di risorse
- Delle aspettative non realistiche
- La mancanza di supporto esecutivo

GESTIONE DI QUALITÀ

È una funzione di più recente introduzione aziendale. La qualità riguarda qui sia i prodotti che i processi e interessa sia committente che la direzione aziendale.
La garanzia di qualità produce confidenza, ma richiede l'applicazione rigorosa dei processi adottati e la loro manutenzione migliorativa (ciclo di PDCA).

I

IMPLEMENTATION

Termine inglese per *codifica*. Si tratta di realizzare concretamente quello che nel progetto è stato precedentemente ideato.

INCAPSULAMENTO

Un meccanismo del linguaggio di programmazione atto a limitare l'accesso diretto agli elementi dell'oggetto.

INCREMENTO

Modo di avvicinarsi sempre di più a destinazione "aggiungendo qualcosa che mi porta verso la meta".

INSPECTION

È un metodo pratico di lettura come Walkthrough. Ha come obiettivo rilevare la presenza di difetti eseguendo una lettura mirata. Chi la fa sono Verificatori distinti dai Programmatori e la strategia che viene adottata si focalizza sulla ricerca con presupposti. In ogni fase delle attività svolte deve avvenire la documentazione:

1. Pianificazione
2. Definizione lista di controllo
3. Lettura
4. Correzione dei difetti

Differenze: Walkthrough richiede maggiore attenzione ed è più collaborativo, mentre Inspection è più rapido e basato su presupposti.

INTEGRAZIONE CONTINUA

In modo dimostrabile incremento e non itero. Vado quindi verso la meta e ogni passo che eseguo ha un valore aggiuntivo.

ITERAZIONE

Procedere per raffinamenti o rivisitazioni "aggiungendo o togliendo qualcosa che mi fa quindi avanzare o retrocedere".

J

JOINT REVIEW PROCESS

Tratta di revisioni di monitoraggio. Si tratta di una revisione **interna** (in particolare RP e RQ).

L

LAZY

Lazy evaluation è una strategia di valutazione che *delays the evaluation of an expression until its value is needed* (non-strict evaluation) e che inoltre evita valutazioni ripetute (sharing).

LEAD TIME

È quanto tempo trascorre da quando un'azione è assegnata a quando questa è completata. Riesco a capire che una persona non sta svolgeno il proprio lavoro se il tempo medio aumenta. Se stiamo finendo invece, il valore tenderà a zero.
È una metrica di project management.

LOGGER

Componente non intrusivo di registrazione dei dati di esecuzione per l'analisi dei risultati, quale per esempio un *bot*.
È uno dei metodi di automatizzazione per i test.

M

MANUALI

Uno dei prodotti che racconta il prodotto all'utente.

MANUALE DELLA QUALITÀ

Il documento che definisce il sistema di gestione della qualità di un'organizzazione. Ha una visione ad alto livello. Si integra con i processi e le procedure aziendali e fissa gli obiettivi di qualità aziendali e le strategie per perseguirli.

MANUTENZIONE

Complesso delle operazioni necessarie a conservare la conveniente funzionalità ed efficienza del prodotto. Può essere:

- **Correttiva** = rimozione di difetti
- **Adattativa** = raffinamento dei requisiti
- **Evolutiva** = evoluzione del sistema

Dato che bisogna avere memoria di quello che ha funzionato in passato e di quello che funziona ora, possiamo dire che un prodotto sotto manutenzione ha una storia, ed essa va gestita con controllo di versione.

MATURITÀ

Misura la qualità dei processi, ovvero misura quanto (e quanto bene) l'azienda è governata dal suo sistema di processi. È quindi caratteristica di un insieme di processi e rappresenta il risultato delle *capability* dei processi considerati.

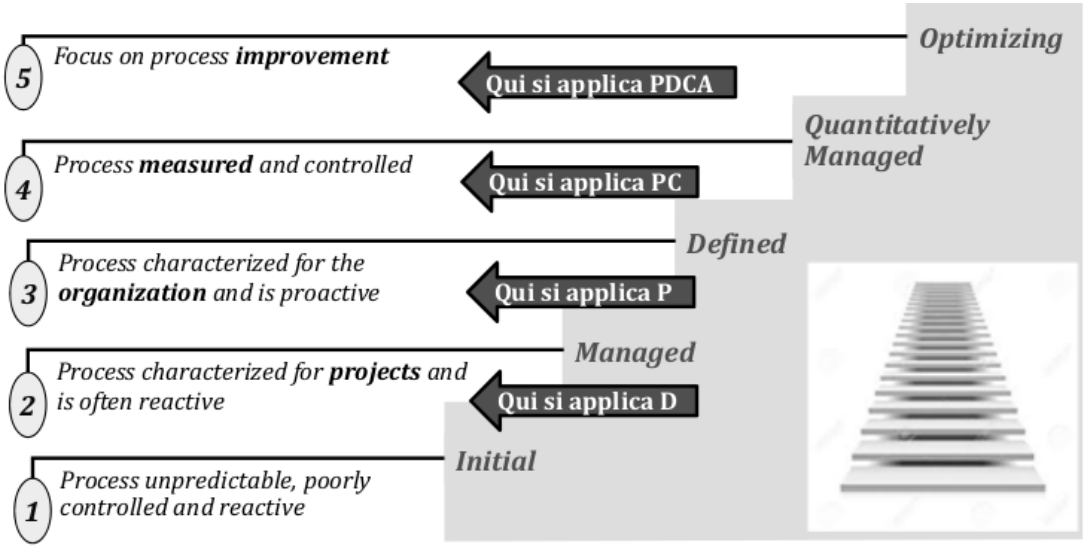


Figura 6: I 5 livelli di maturità per la qualità di processo.

I livelli di maturità di CMMI mi aiutano a capire con quale intelligenza agisco.
La maturità di prodotto valuta il grado di evoluzione del prodotto:

- Quanto migliora in seguito alle prove
- Quanto diminuisce la densità dei difetti
- Quanto può costare la scoperta del prossimo difetto

METRICA

Integrale degli usi o utenti nel tempo.
La *Metrica Software* comprende:

- **SLOC** (= *Source Lines Of Code*): conta le linee, è quindi oggettivamente misurabile e dà limiti
- **Effort**: risorse umane misurate come giorni/persona
- **Testo**: perché il testo è facilmente offuscabile, quindi si usa "l'indice di nebbia" (o di leggibilità) ovvero

```
Fog = ((average number of words / sentence) + (number of words of 3
                    syllables or more)) * 0.4
```

Dà il modo di classificare attributi di processo e ci aiuta a ragionare a monte del problema e non a valle. In questo modo si possono predire gli attributi che arriveranno (in parole povere, capire prima se il prodotto farà schifo). L'obiettivo è quello di identificare anomalie *"the sooner the better"*.
Associate alla qualità del software c'è l'idea di *assunzioni* sulle metriche: il nostro modo di pensare al problema.
Attenzione perché non sempre si può misurare ciò che vogliamo. Ci interessa misurare ciò che è tracciabile di quello che vogliamo sapere (esempio dell'analisi del sangue: voglio sapere una cosa di alto livello non misurabile, tramite dati del mio sangue che sono misurabili). Importanti *attributi* sono:

- Manutenibilità
- Affidabilità
- Portabilità
- Usabilità

Il problema delle metriche SW è che il software è immateriale, difficile da misurare e le tecnologie SW cambiano rapidamente. Una metrica di riferimento per la misurazione del software è il lead time.
Metriche di progettazione:

- **Grado di coesione**: conviene che le parti siano altamente coese tra di loro.
- **Grado di accoppiamento**: questo se alto è dannoso perchè la modifica in uno crea danni sull'altro.

- **Complessità strutturale**: dipendenza da fuori (esempio del metodo con 30 parametri). Funzione del *fan-out* di una porta logica (è il numero di porte logiche che possono essere collegate alla sua uscita).
- **Complessità del flusso dati**: funzione del numero di parametri in ingresso e in uscita.
- **Complessità del sistema**: funzione di complessità strutturale e complessità del flusso di dati.

Inoltre, la *complessità ciclomatica* ci interessa perchè più cammini ho, più test devo fare.

MILESTONE

La milestone indica un punto nel tempo associato ad un valore strategico. Ogni milestone di calendario è associata a uno specifico insieme di baseline. Ogni milestone dev'essere:

- Specifica
- Raggiungibile
- Misurabile (per quantità d'impegno necessario)
- Traducibile in compiti assegnabili
- Dimostrabile agli stakeholder

Le milestone sono un elemento essenziale dei modelli di sviluppo. C'entrano infatti sia con obiettivi, che con rischi e vincoli.
Quante milestone ho lo decide il fornitore. La baseline invece sono le evidenze che porto per aprire i gate. Prima vengono le milestone e poi le baseline.

MISTAKE

Ha a che vedere con la gerarchia di problemi nei test.
Ci siamo sbagliati noi nel produrre qualcosa che, se utilizzato nel SW, causerà errore. Questo è quindi a monte di tutta la gerarchia: il primo elemento che causa il fallimento.

MISURA

La misura assegna un valore quantitativo a un'entità per caratterizzarne un attributo specifico.
Obiettivo: rendere oggettivi i risultati delle valutazioni effettuate, perchè siamo interessati in qualcosa di osservabile che ci aiuti a capire come siamo messi.
Quest'oggettività implica:

- Ripetibilità
- Confrontabilità
- Confidenza

Nei limiti dell'approssimazione e della possibilità di astrazione.
Per attuare queste misurazioni abbiamo due modi. **misura di realtà** e **indicatore**. La misura di realtà è diversa dall'indicatore. La misura di realtà è per esempio il metro (per l'altezza). Gli indicatori invece ci servono per un confronto, mi suggerisoco qualcosa, ma sono solo degli indicatori: non danno delle verità.
Un giudizio di valutazione su queste misurazioni si forma con:

- **Serie storica**: diagramma che ci dice l'andamento (esempio della polizia Svizzera che mette la multa all'auto dopo che ha percorso tutto lo stato con velocità in eccesso). Dà delle tendenze nel tempo che sono molto utili.
- **Cruscotto**: alimentato in push, ovvero sempre valido, aggiornato e sempre a nostra disposizione. Ci deve dare informazioni.

MOCK

I *mock object* sono degli oggetti simulati che riproducono il comportamento degli oggetti reali in modo controllato. Nella programmazione ad oggetti, un programmatore crea un oggetto mock per testare il comportamento di altri oggetti, reali, ma legati ad un oggetto inaccessibile o non implementato. Allora quest'ultimo verrà sostituito da un mock.
Questi oggetti sono utili qualora io debba fare test di unità.

MODELLI DI SVILUPPO

Il modello è una costruzione astratta che fa capire qual è il problema e dà strumenti per risolverlo. È un riferimento ideale ad una cosa concreta, quindi astrazione non eseguibile, ma che mi definisce le proprietà. Date le diverse transizioni previste tra gli stati di un ciclo di vita e diverse regole di attivazione, esistono diversi tipi di modelli. I modelli, per essere tali, sono tutti "buoni". Semplicemente alcuni sono più adatti a certe esigenze rispetto ad altri.

- **A cosa serve**
  Tre cose hanno fortissima influenza sulla scelta da fare:
  - Obiettivi
  - Rischi
  - Vincoli

  Serve a perseguire gli *obiettivi* cercando di rispettare i *vincoli* e mitigando i *rischi*.
- **Di cosa è fatto**
  Il modello di sviluppo è legato al progetto da portare a termine e il progetto determina i processi. Quindi i processi hanno a che vedere con il modello di sviluppo. C'è un importante ordine di attività. Chi decide quali sono i *gate* (momento in cui si termina un'attività e si passa a quella successiva) siamo noi, ovvero il fornitore, in base a quali obiettivi raggiungere nel tempo. Le milestone coincidono con i nostri gate.
  Per dare elementi di mitigazione ai rischi ho la Technology Baseline. Il SEMAT mi dà l'essenza che sta alla radice di ogni modello di sviluppo. Dal SEMAT riesco a capire quali milestone adottare nel mio piano e tra quelle che posso scegliere, le

milestone che scelgo devono essere una confluenza di quelle più specializzate (vedere le cards da immagine). Ho tante baseline quanti "lucchetti"(milestone) da aprire.

- **Come lo si attiva**
  Individuando le attività riconoscibili dalle baseline che ho scelto, collocandole nel tempo disponibile e assegnandole.

Sequenziale (o a cascata)

In breve: ha rigide fasi sequenziali.
Tutto deve essere ripetibile (quindi anche migliorabile) e lineare (una successione di fasi), ma non ammette un ritorno a fasi precedenti. Adotta quindi una strategia in cui si continua o al più ci si ferma. Si prosegue solo se l'azione viene considerata buona (quindi ben documentata prima). I suoi prodotti sono principalmente *documenti*.
Ogni fase è caratterizzata da pre e post condizioni (di ingresso le prime e di uscita le seconde) e viene definita in termini di responsabilità e ruoli coinvolti. Dato che le fasi sono durate temporali, devono essere distinte e non sovrapposte nel tempo, dato che presentano delle dipendenze causali tra loro.
Lo schema per il modello a cascata prevede in ordine:
Analisi - Progettazione - Realizzazione - Manutenzione.

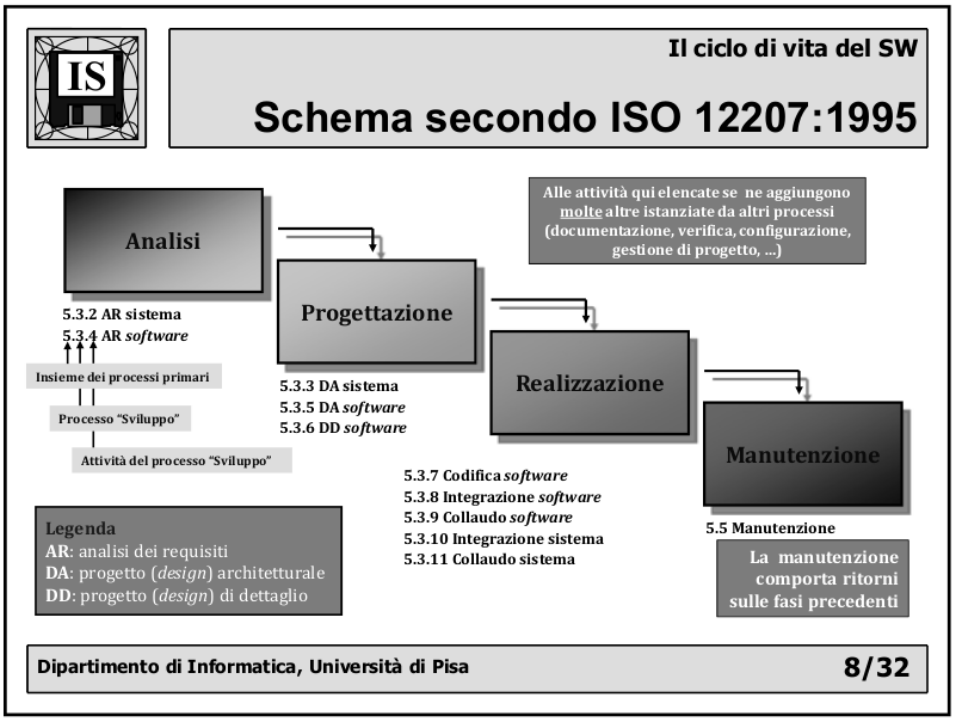


Figura 7: Schema del modello a cascata secondo lo standard ISO 12207.

Difetto: è totalmente rigido quindi ha bisogno di correttivi (come i prototipi).

Incrementale

In breve: realizzazione in più passi. Prevede rilasci multipli e successivi in cui ciascuno realizza un incremento di funzionalità.

Dato che spesso non conviene posticipare l'integrazione di tutte le parti del sistema, risulta migliore l'integrazione continua di piccole parti. Sceglie un ordine di sviluppo che prepari di volta in volta il passaggio successivo, motivo per cui ci vuole un modo strategico per capire come muoversi.
I primi incrementi possono essere frutto di prototipazione, aiutando a fissare meglio i requisiti per gli incrementi successivi. Questi primi incrementi puntano a soddisfare i requisiti più importanti sul piano strategico, così essi diventano presto chiari e stabili, e quelli meno importanti si possono armonizzare con lo stato del sistema.
In questo modo *ogni incremento riduce il rischio di fallimento.*
Analisi e progettazione architetturale non vengono ripetute, dato che l'architettura del sistema è ben identificata e fissata dall'inizio, invece la realizzazione può presentare dei ritorni (per esempio, al momento della validazione può avvenire un ritorno).

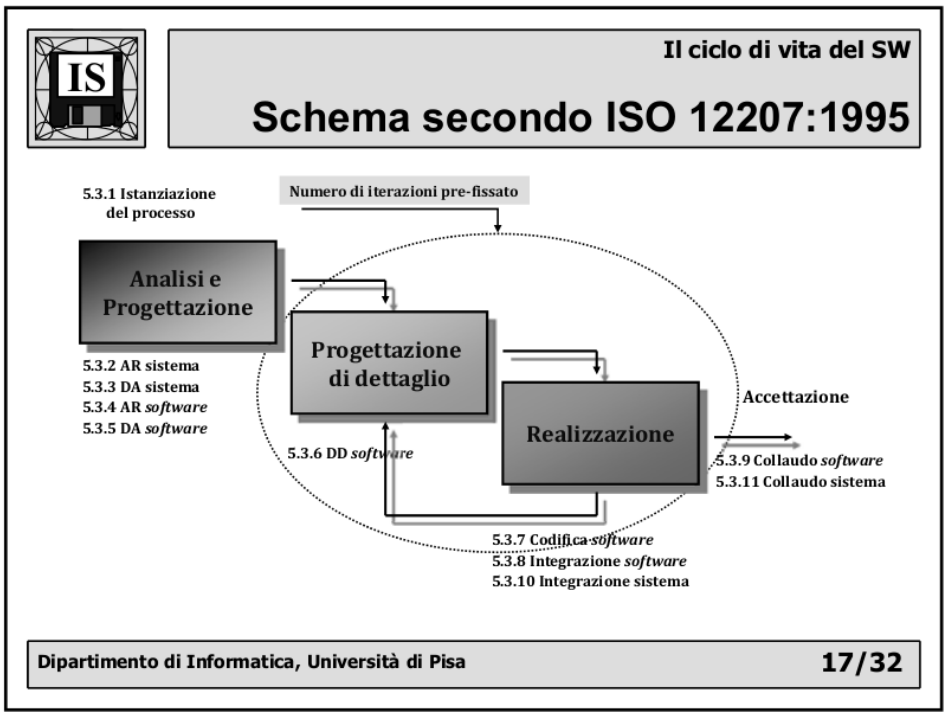


Figura 8: Schema del modello incrementale secondo lo standard ISO 12207.

Iterativo

In breve: ha ripetute iterazioni interne.
Questo tipo di modello è applicabile a qualunque modello di ciclo di vita, consente infatti maggior capacità di adattamento.
Difetto: il problema è che può andare incontro al rischio di non convergenza perché ogni iterazione comporta un ritorno all'indietro nella direzione opposta all'avanzamento del tempo.
In generale conviene quindi decomporre la realizzazione del sistema in parti più piccole trattando prima le parti più critiche (magari quelle i cui requisiti vanno maggiormente chiariti).

A evoluzioni successive

In breve: comporta il riattraversamento di più stati di ciclo di vita.
Prodotto che evolve come conseguenza di una manutenzione. Aiuta a rispondere a bisogni non inizialmente preventivabili.
Inizialmente viene fatta un'analisi preliminare che identifica i requisiti, definisce l'architettura di massima e pianifica i passi di analisi. Dopodiché avviene l'analisi e realizzazione di una singola evoluzione, per raffinamento dell'analisi iniziale o per progettazione-codifica-prove-integrazione. Infine c'è il rilascio di versioni che man mano saranno sempre più complete.
Ogni fase ammette iterazioni multiple e parallele.

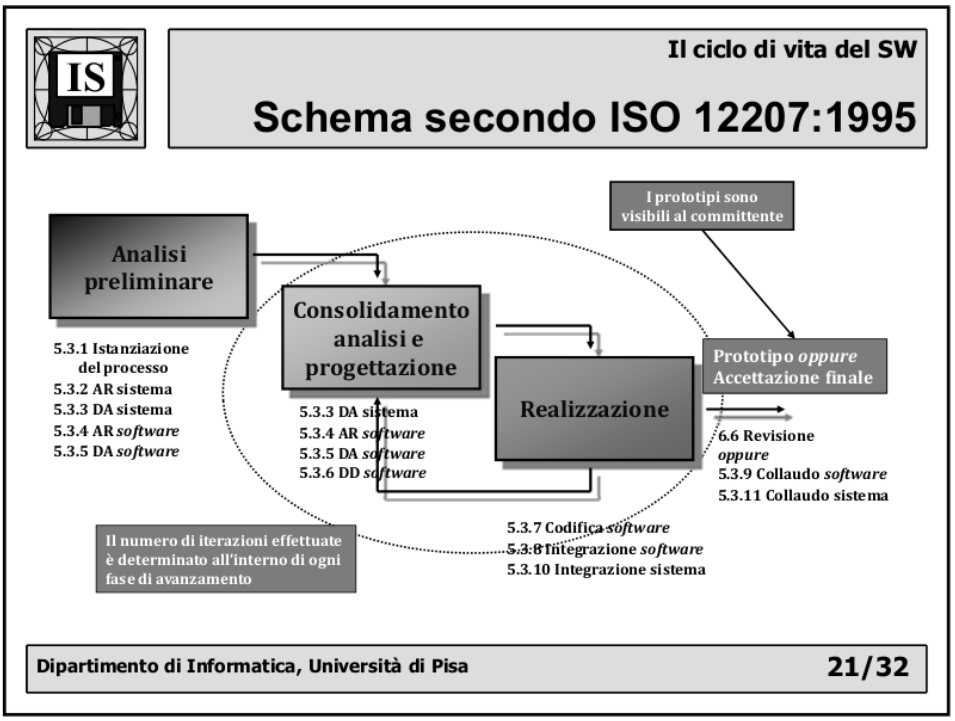


Figura 9: Schema del modello evolutivo secondo lo standard ISO 12207.

Per componenti

In breve: è orientato a massimizzare il riuso di software.
Il prodotto viene decomposto in parti già esistenti e funzionanti, riusabili. L'analisi dei requisiti viene in seguito rivisitata in base alle possibilità di riuso.

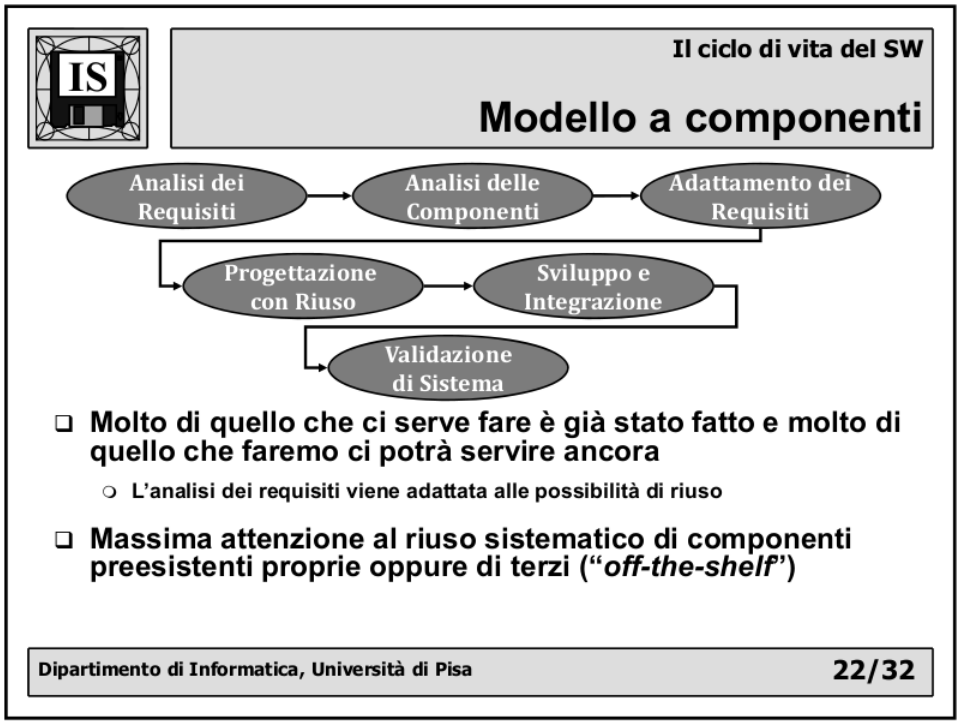


Figura 10: Schema del modello a componenti secondo lo standard ISO 12207.

Il riuso (dal Sommerville)
Le unità software riusate possono avere diversa dimensione, per esempio:

- **System reuse**: un intero sistema (che può essere composto da più applicazioni) può essere riusato come parte di un grande sistema contenente tanti sistemi
- **Application reuse**: un'applicazione può essere riusata incorporandola in altri sistemi senza fare cambiamenti, oppure configurando l'applicazione per diversi customers
- **Component reuse**: componenti di un'applicazione (da sotto-sistemi a singoli oggetti) possono essere riusate. Per esempio, un sistema di sviluppo che fa pattern-matching, che fa parte di in un sistema di text-processing, può essere riusato in un sistema di gestione di database. Le componenti possono risiedere in un cloud o in server privato e possono essere accessibili tramite *Application Programming Interface* (API).
- **Object and function reuse**: componenti software che implementano una singola funzione o una classe oggetto possono essere riusate. Molte librerie di funzioni e classi sono liberamente disponibili. Si riusano classi e funzioni in queste librerie collegandole con lo sviluppo di nuovo codice.

A volte i sistemi o componenti sono così specifici che è molto costoso modificarli per una nuova situazione. Quindi, più che riusare il codice, è possibile riusare le idee che stanno alla base del software. Questo è chiamato **concept reuse** e tratta per esempio di riusare un way of working o un algoritmo, e approcci quali i design patterns.
I *benefici* del riuso sono:

- Costo complessivo di sviluppo più basso, perché meno componenti software devono essere progettate, implementate e validate

- Sviluppo accelerato
- Aumento di affidabilità, perchè software che è stato provato e testato in altri sistemi funzionanti è più affidabile di un nuovo software. I suoi difetti di progettazione e implementazione dovrebbero già esser stati trovati e sistemati.
- Ridotto rischio di processo, vero specialmente per grandi componenti software riusate come sottosistemi. È un fattore importante per il Project Manager perché riduce il margine di errore nella stima dei costi di un progetto.
- Conformità con gli standard, perché alcuni standard, come gli interface standard, possono essere implementati come set di componenti riusabili

Ci sono però anche delle *difficoltà* legate al riuso come:

- Costo associato al capire se una componente è adatta al riuso per una particolare situazione o meno. È un costo aggiuntivo che potrebbe essere così elevato che potrebbe non abbassare il costo complessivo di sviluppo
- Creare, mantenere e usare la componente di una libreria, perché può essere oneroso
- Maggiori costi di manutenzione, perché se il codice sorgente della componente riusata non è disponibile, gli elementi riusati del sistema potrebbero diventare incompatibili con i cambiamenti fatti al sistema
- Mancanza di strumenti di supporto, perché alcuni tools non supportano lo sviluppo con riuso
- *"Not-invented-here" syndrome*, ovvero il fatto che alcuni software engineers preferiscono riscrivere componenti perchè credono di poterle migliorare

Fattori da considerare quando si sta pianificando il riuso:

- **The development schedule for the software**: se il software deve essere sviluppato velocemente, bisognerebbe riusare completamente un sistema piuttosto che una componente singola.
- **The expected software lifetime**: se si pensa ad un sistema con ciclo di vita lungo, bisognerebbe concentrarsi sulla manutenibilità del sistema. In questo caso si mira a riusare sistemi e componenti open-source più sicuri.
- **The background, skills and experience of the development team**: la cosa migliore è focalizzarsi nel riuso di aree in cui il team ha esperienza.
- **The criticality of the software and its non-functional requirements**: per un sistema che deve essere certificato da un regolatore esterno, bisognerebbe creare una parte di sicurezza, ma questo è difficile se non si ha accesso al codice sorgente. Per esempio, se il software ha dei requisiti stringenti, potrebbe risultare impossibile usare strategie quali il *model-driven engineering* (MDE).
- **The application domain**: in molti domini applicativi, come sistemi industriali e d'informazione medica, ci sono prodotti generici che possono essere riusati in una situazione locale. Qui fare riuso è spesso conveniente.

- **The platform on which the system will run**: come i .NET sono specifici delle piattaforme Microsoft, esistono sistemi di applicazioni generici che sono specifici di una certa piattaforma. Si è quindi in grado di riusare solo se il nostro sistema è progettato per la stessa piattaforma.

In questo contesto, il framework è un ottimo esempio di riuso.
Quando una compagnia deve supportare un numero di sistemi simili ma non identici, uno degli approcci di riuso più efficaci è creare una *software product line*. Una *software product line* è un set di applicazioni con un'architettura comune e componenti condivise in cui ogni applicazione è specializzata per riflettere specifici requisiti del cliente. Generalmente un'applicazione di base include:

- **Core components**: forniscono un'infrastruttura di supporto e di solito non vengono modificate quando viene sviluppata una nuova istanza della *software product line*.
- **Configurable application components**: possono essere modificate e configurate per specializzare una nuova applicazione.
- **Specialized application components**: specifiche del dominio, alcune o tutte possono essere rimpiazzate quando una nuova istanza della *software product line* viene creata.

CBSE Le componenti sono ad un livello di astrazione più alto degli oggetti e sono definite dalle loro interfacce. Sono solitamente più grandi degli oggetti singoli e tutti i dettagli implementativi sono nascosti alle altre componenti.
Component-based software engineering è il processo di definizione, implementazione e integrazione o composizione di queste componenti scarsamente accoppiate e indipendenti nei sistemi.
Le caratteristiche essenziali di una componente, usate nel CBSE (*Component-based software engineering*), sono:

- **Composable**
- **Deployable**
- **Documented**
- **Independent**
- **Standardized**

Un modo utile di vedere una componente è vederla come un fornitore di uno o più servizi:

- La *componente* è un'entità eseguibile indipendente che viene definita dalla sua interfaccia. Non c'è bisogno di saperne il codice sorgente per usarla.
- I *servizi* offerti da una componente sono resi disponibili attraverso un'interfaccia e tutte le interazioni avvengono tramite quell'interfaccia. L'interfaccia della componente è espressa in termini di operazioni parametrizzate e il suo stato interno non è mai esposto.

Di base tutte le componenti hanno due interfacce collegate:

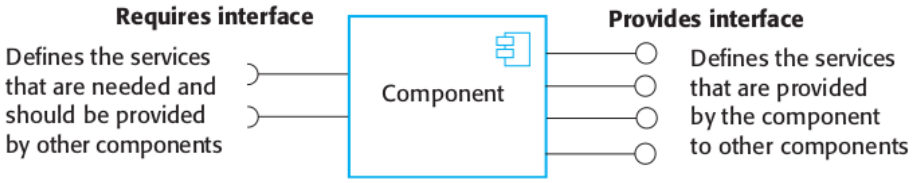


Figura 11: Interfacce di una componente.

- La *requires interface* specifica i servizi che le altre componenti del sistema devono fornire affinchè la componente funzioni correttamente. Se questi servizi non sono disponibili, la componente non funzionerà.
- La *provides interface* definisce i servizi forniti dalla componente. Quest'interfaccia è la componente API.

Component models Un *component model* è una definizione di standard per l'implementazione, la documentazione e il deployment di componenti. Questi standard sono per gli sviluppatori delle componenti e assicurano che le componenti siano interoperabili. Per *service components* il più importante modello a componenti è il Web Service model, mentre per le *embedded components* sono ampiamenti usati i modelli Enterprise Java Beans(EJB) e Microsoft's .NET.

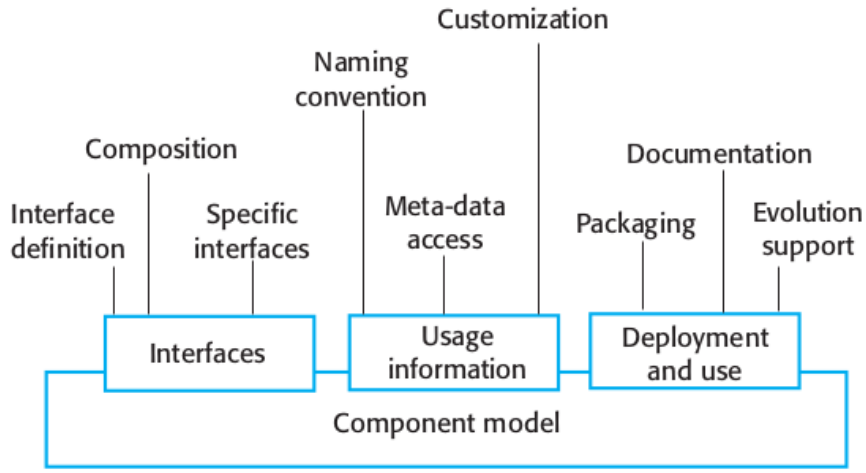


Figura 12: Elementi base di un component model.

CBSE processes I processi CBSE sono processi software che supportano la *Component-Based Software Engineering*. Ne esistono di due tipi:

- **Development for reuse**: processo mirante allo sviluppo di componenti o servizi che saranno riusati in altre applicazioni. Generalmente coinvolge componenti generiche già esistenti.
- **Development with reuse**: processo che sviluppa nuove applicazioni usando componenti e servizi esistenti.

Agile

In breve: è altamente dinamico ed è fatto di brevi ciclo iterativi e incrementali.
Basato su principi fortemente di reazione che sono:

1. **Individuals and interactions over processes and tools**: l'eccessiva rigidità ostacola l'emergere del valore
2. **Working sofware over comprehensive documentation**: la documentazione non sempre corrisponde a SW funzionante
3. **Customer collaboration over contract negotiation**: l'interazione con gli stakeholder va incentivata
4. **Responding to change over following a plan**: la capacità di adattamento al cambiare delle situazioni

Si basa sull'idea di *user story*, ovvero una funzionalità che è significativa per l'utente e che vuole nella realizzazione del software richiesto. Ogni *user story* è definita in un documento che descrive il problema, una sintesi delle conversazioni in cui si è discusso del problema con gli stakeholder e la strategia da adottare per soddisfare il problema. Si prosegue suddividendo il lavoro in piccoli incrementi (che possono essere sviluppati anche indipendentemente) sviluppati in modo continuo e sequenziale dall'analisi all'integrazione. Gli *obiettivi strategici* sono poter dimostrare costantemente al cliente quanto fatto e che l'intero prodotto SW sia ben integrato e verificato.
Difetto: non viene considerato molto buono perché è assente la documentazione, quindi è più un costo che un valore.

SCRUM

In breve: è un tipo di modello agile ("mischia di rugby").
È costituito da:

- **Product backlog**: requisiti e funzionalità del prodotto
- **Sprint**: fase operativa di sviluppo
- **Sprint backlog**: insieme di storie del prossimo sprint
- **Sprint Planning**: pianificazione dello sprint
- **Daily Scrum**: controllo giornaliero dell'avanzamento
- **Sprint Review**: controllo prodotti dello sprint
- **Sprint Retrospective**: controllo qualità sullo sprint

A spirale

(Secondo il prof: "Complicato, non lo trattiamo"). Va da un problema piccolo ad uno sempre più grande. È pensato per attività che non hanno una best practice ben nota.

MODULI

La più piccola entità progettuale che sia utile rappresentare.

N

NORME DI PROGETTO

È un documento **interno** che serve a istituire un way of woking, ovvero come ci impegniamo a lavorare. È per noi un documento incrementale, perché aggiungo regole per le attività man mano che ne ho bisogno, e deve essere leggero, cioè facile da consultare e da modificare. Le Norme di Progetto tra le verie cose includono come programmare: nello specifico, il processo di sviluppo contiene le norme di codifica.

O

OPPORTUNITY

Bisogno di un cliente che vale la pena di soddisfare.
È uno degli elementi di un progetto (vedi immagine).

ORACOLO

È un elemento di una prova.
Colui che risponde a tutti i dubbi e domande. È un metodo per generare a priori i risultati attesi e per convalidare i risultati ottenuti nella prova. Generalmente viene applicato da agenti automatici, per velocizzare la convalida e renderla "oggettiva".
Si producono oracoli sulla base di specifiche funzionali con prove semplici tramite l'uso di componenti di terze indipendenti.

P

PIANIFICAZIONE

Definisce delle attività, al fine di pianificarne lo svolgimento e controllarne l'attuazione. Serve come base su cui gestire l'allocazione delle risorse e per stimare e controllare scadenze e costi.
Per la pianificazione vengono utilizzati degli *strumenti* quali: Diagramma di Gantt, PERT e WBS.
In breve: chi fa cosa - quando - in quanto tempo - quanto costa.

PIANO DI PROGETTO

Il piano di progetto (o *PdP*) sostanzialmente è un documento ufficiale (di tipo esterno), soggetto ad approvazioni, con il quale si descrivono gli obiettivi di progetto e gli elementi necessari per il loro raggiungimento. Si vedono le risorse disponibili e le loro assegnazioni alle attività con scansione delle attività nel tempo.
Principalmente è scomposto in:

- Pianificazione (preventiva)
  - Team
  - Analisi dei rischi
- Consuntivazione: è lo specchio della pianificazione. Fa una valutazione retrospettiva, ovvero guardando all'indietro quello che si è fatto.

Tipicamente la *struttura* del PdP è composta da:

- Introduzione (scopo e struttura)
- Organizzazione del progetto
- Analisi dei rischi
- Risorse disponibili (tempo e persone)
- Suddivisione del lavoro (*work breakdown*)
- Calendario delle attività (*project schedule*)
- Meccanismo di controllo e di rendicontazione

È da tenere ben presente che l'analisi dei rischi è l'attività più importante. Questo perchè ci sono anche dei rischi di progetto che derivano da *fonti di rischio* quali:

- Tecnologie di lavoro e produzione
- Rapporti interpersonali
- Organizzazione del lavoro
- Rapporti con gli stakeholder
- Tempi e costi

Per questo ci vuole una Gestione dei rischi.

PIANO DI QUALIFICA

Attività del sistema qualità mirante a fissare le politiche aziendali e determinare gli obiettivi di qualità del singolo progetto, i processi e le risorse necessarie per conseguirli.
Definisce le strategie di verifica e validazione (metodi, tecniche, procedure, strumenti e tempi): stabilisce quindi qual è il modo in cui noi perseguiamo la qualità. Si parla qui di metriche e obiettivi "di oggi" (ad esempio requisiti, fluttuazione requisiti). Per ogni passo fatto in analisi o altro, è necessario preparare i test e fare il tracciamento per non dimenticarsi nulla, al fine di dare qualità (consultare modello a V). In esso inoltre si specificano quali e quante prove effettuare.
Anche questo documento, come le Norme, è incrementale (ma di tipo esterno).
N.B: le metriche vanno inserite nelle Norme, mentre gli obiettivi nel PdQ.

PPP

Stanno per: *Product*, *Process*, *Progression in learning* (e *Incrementalmente*). Sono i 3 (4) punti di avanzamento chiave del progetto. Le 3P sono la *capstone* (=parte finale di una struttura) *project*, ovvero un assignment che di solito avviene nell'ultimo anno accademico o di scuola superiore di uno studente, in cui lo studente applica ciò che ha imparato a una situazione reale.

PORTABILITÀ

Un prodotto che ha la proprietà di adattarsi ad altri dispositivi.

PREVENTIVO

Stima iniziale dei costi necessari alla realizzazione di un progetto. Il *PAF* (= preventivo a finire) è da attuarsi in corso d'opera perché deve diventare sempre più preciso col passare del tempo. Questo perché, naturalmente, dopo una revisione, se qualche costo è cambiato, bisogna aggiornare il costo che si prevede per la fine.
NB: il preventivo non si può mai aumentare, ma solo decrementare (la proponente non è contenta di pagare più di quello che si era inizialmente stimato).

PRINCIPIO DEL MIGLIORAMENTO CONTINUO

Identifica specifici obiettivi di miglioramento, li esegue, ne verifica l'esito e infine agisce sulle conseguenze (per esempio, se considerato buono, viene tenuto nello standard).

PROBLEM RESOLUTION

Si tratta di avere un metodo molto chiaro, non violabile, per la soluzione di problemi. Proviene dallo standard ISO 12207 ed è una tecnica di verifica.
È fatto di una strategia nota (il nostro way of working). In essa tutti i problemi sorti sono stati registrati e ogni problema va studiato individualmente, decidendo per esso quali soluzioni potrebbero essere accettabili. Viene in seguito realizzata la soluzione scelta e verificato l'esito della correzione.
Questa comunque è un'azione correttiva e non buona, fatta tramite Test di regressione.

PROCESSO

Insieme di attività correlate e coese che trasformano bisogni in prodotti (qualunque risultato di processo si chiama prodotto). Opera secondo regole e nel farlo consuma risorse. Questo insieme di attività (tra cui la Gestione di Progetto) deve essere efficiente ed efficace. Adottare standard di processo ci aiuta a raggiungere l'economicità.
Tra le *attività di processo* per lo sviluppo software troviamo:

1. **Istanziazione del processo**
2. **Analisi dei requisiti del sistema**
3. **Progettazione architetturale del sistema**
4. **Analisi dei requisiti del SW**
5. **Progettazione architetturale del SW**
6. **Progettazione di dettaglio del SW**
7. **Codifica e prova delle componenti SW**: comprende i compiti:
   - Definire procedure e dati di prova
   - Eseguire e documentare le prove
   - Aggiornare documentazione e pianificare prove d'integrazione
   - Valutare l'esito delle prove
8. **Integrazione delle componenti SW**: comprende i compiti:
   - Definire il piano di integrazione
   - Eseguire e documentare le prove
   - Aggiornare documentazione e pianificare prove di collaudo
   - Valutare l'esito delle prove
9. **Collaudo del SW**: comprende i compiti:
   - Eseguire e documentare il collaudo
   - Valutare l'esito del collaudo
10. **Integrazione di sistema**: comprende i compiti:
    - Eseguire e documentare le prove
    - Aggiornare documentazione e pianificare prove di collaudo
    - Valutare l'esito delle prove
11. **Collaudo del sistema**: comprende i compiti:
    - Eseguire e documentare il collaudo
    - Valutare l'esito del collaudo

L'adozione di processi deve essere consapevole ed efficace, per questo ci deve essere una buona *organizzazione*:

- **Processo standard**: è il riferimento di base generico ed è condiviso tra diverse aziende nello stesso dominio applicativo.
- **Processo definito**: è una specializzazione del processo standard al fine di adattarlo a specifiche esigenze.
  Tra i processi definiti troviamo:
  - **Processi specializzati per azienda**: sono chiari, stabili, documentati, indipendenti dal modello di ciclo di vita adottato, dalle tecnologie, dal dominio applicativo e dalla documentazione richiesta
  - **Processo di progetto**: è un'istanziazione di processi definiti che utilizzano risorse aziendali per raggiungere obiettivi prefissati e con scadenze (progetti). I processi specializzati per progetto sono ben pianificati, hanno chiare scelte di specializzazione e l'esito viene valutato in maniera critica.

L'organizzazione interna dei processi, per essere buona, deve inoltre essere incentrata sul principio di miglioramento continuo: il ciclo di Deming.
Per quel che riguarda la specializzazione di processi, abbiamo dei *fattori di specializzazione* che riguardano:

- Dimensione e complessità del progetto
- Rischi come dominio applicativo e tecnologie in uso
- Competenza delle risorse umane
- Fattori dipendenti dal contratto

Per ogni specializzazione bisogna definire lo scenario di applicazione e le attività, e organizzare le relazioni tra processi.
Fattori d'influenza:

- I processi non determinano quale ciclo di vita, ma al contrario, il ciclo di vita determina quali processi attivare
- Il livello di coinvolgimento del cliente determina le revisioni necessarie e l'intensità della documentazione
- Il ciclo di vita viene scelto in base a cosa vuole il committente (se versione unica non modificabile o versione destinata a continue evoluzioni) e al tipo di coinvolgimento del committente nell'accertamento dello stato di avanzamento (se revisioni interne o esterne, bloccanti o non bloccanti)

PRODOTTO SOFTWARE

Software progettato per essere rilasciato all'utente. Viene visto come insieme di parti (che hanno a che vedere con forma, contenuto e funzione) organizzate secondo una configurazione per ragioni di convenienza e semplicità. La forma che ha dipende dalla richiesta e può essere:

- **Commessa**: se le parti vengono fissate del committente
- **Pacchetto**: se le parti sono idonee alla replicazione
- **Componente**: se le parti sono adatte alla composizione
- **Servizio**: se le parti sono fissate dal problema

Gli stati principali di un prodotto SW sono:

- Concezione
- Sviluppo
- Utilizzo
- Ritiro

Noi, nel progetto, ci concentriamo sullo sviluppo, per questo parliamo di modelli di sviluppo.
Il suo comportamento viene delineato dal ciclo di vita che, se lungo, ha notevoli costi di manutenzione. Il prodotto deve essere *configuration item*: stare su una repository.
Il software deve fare queste cose quantificabili:

- **Cosa**: il software deve fare quei calcoli (caratteristiche funzionali)
- **Come**: in che modo fare quei calcoli (caratteristiche non funzionali)

e ciò richiede verifica.

PRODUCT BASELINE

Presenta la baseline architetturale del prodotto (design patterns adottati) e va data tramite "allegato tecnico" (una banale cartella compressa per mail) con diagrammi delle classi e di sequenza. In questo momento deve esistere un prodotto, idealmente ""finito"", e bisogna essere pronti alla validazione.
È parte della RQ.

PRODUTTIVITÀ

Misurazione dell'efficienza in base alle aspettative soddisfatte.

PROGETTAZIONE

In inglese *Design*.
Essenzialmente si tratta di pensare ad una soluzione (*Come fare la cosa giusta?*) dopo aver fatto l'Analisi dei Requisiti (*Qual è il problema e qual è la cosa giusta da fare?*). Essa precede la realizzazione perseguendo la correttezza per costruzione anziché la correttezza per correzione.
È importante progettare per:

- Dominare la complessità del prodotto

- Organizzare e ripartire le responsabilità di realizzazione
- Produrre in economia (efficienza)
- Garantire qualità (efficacia)

Mentre l'analisi richiede comprensione del dominio e attua un *approccio investigativo* (da uno a tanti), la progettazione ricerca una soluzione soddisfacente per tutti gli stakeholder, descrive l'architettura del prodotto prima di pensare al codice e attua un *approccio sintetico* (dal tanto al poco).
È un'attività molto parallelizzabile.
Gli obiettivi della progettazione sono:

- Soddisfare i requisiti con un sistema di qualità definendo l'architettura logica del prodotto (impiegando parti con specifica chiara e coesa realizzabili con risorse sostenibili e costi fissati, organizzate in modo da facilitarne cambiamenti in futuro)
- Ricercare soluzioni architetturali utili al caso e con parti riusabili
- Dominare la complessità del sistema suddividendolo in parti (divide-et-impera) con complessità trattabile e in modo da essere un compito rapido e verificabile da un singolo individuo
- Spingere quindi la progettazione nel dettaglio ma senza arrivare al momento in cui il costo di coordinamento delle parti supera il beneficio della suddivisione

Progettare per riuso è difficile, come lo è anticipare i bisogni futuri, e non è immediato perchè bisogna minimizzare le modifiche alle componenti riusate, affinché esse non perdano valore. Nel breve periodo il riuso è solo costo. Diventa risparmio solo nel medio termine.
La **progettazione architetturale** può seguire un approccio top-down, bottom-up oppure *meet-in-the-middle* che è una via di mezzo e il più frequentemente usato. Se si impone uno stile architetturale, il *framework* (ha un'architettura) è top-down oltre che contemporaneamente bottom-up dato che è fatto di codice già sviluppato (esempi Spring, Struts, ecc).
Nella progettazione si parla anche di *Design pattern* architetturali. Sono nati con dei principi. Sono la soluzione progettuale a dei problemi ricorrente perché suggeriscono un'organizzazione architetturale con proprietà note, ottenibili solo con una buona istanziazione e coerente implementazione (sono il corrispondente architetturale degli algoritmi).
Gli stati di progresso della progettazione secondo SEMAT sono:

- Architecture selected
- Demonstrable
- Usable
- Ready

La **progettazione di dettaglio** si occupa di definire le unità realizzative in modo che il carico di lavoro sia compiuto dal singolo programmatore.
La progettazione determina il codice che avrà delle qualità, e il codice è buono se ha le qualità che, il pattern che ho scelto, propone. È importante correlare la progettazione

al software, non scinderli! L'approccio migliore sarebbe prima progettare qualcosina di piccolo e poi provare a scrivere il codice.
La progettazione guida la programmazione.

PROGETTISTA

È uno dei ruoli in un progetto. Generalmente sono pochi e accompagnano lo sviluppo, ma non la manutenzione. Hanno competenze tecniche e tecnologie aggiornate, motivo per cui influiscono su quest'ultime.

PROGETTO

Insieme ordinato di 4 attività:

1. **Pianificazione**: gestisce le risorse (come tempo, soldi, ecc..) e le responsabilità. Non è un'attività parallelizzabile e la ripianificazione va fatta `n` volte (con `n` a piacere).
2. **Analisi dei requisiti**: definisce **cosa** fare.
3. **Progettazione**: definisce **come** fare. È molto emancipante.
4. **Realizzazione**: esegue perseguendo qualità e fa verifica e validazione.

Queste attività realizzano processi di ciclo di vita e devono:

- Raggiungere obiettivi
- Avere inizio e fine fissate (scadenze)
- Avere risorse limitate
- Consumare le suddette risorse

Nel suo complesso il progetto è sempre *collaborativo*. L'obiettivo del progetto non è fare, ma capire cosa fare prima di agire. Questo per emanciparsi.
La disciplina da prendere come riferimento è l'Ingegneria del Software.
L'organizzazione SEMAT definisce i principali elementi di un progetto:

- **Customer**
- **Solution**
- **Endeavor**

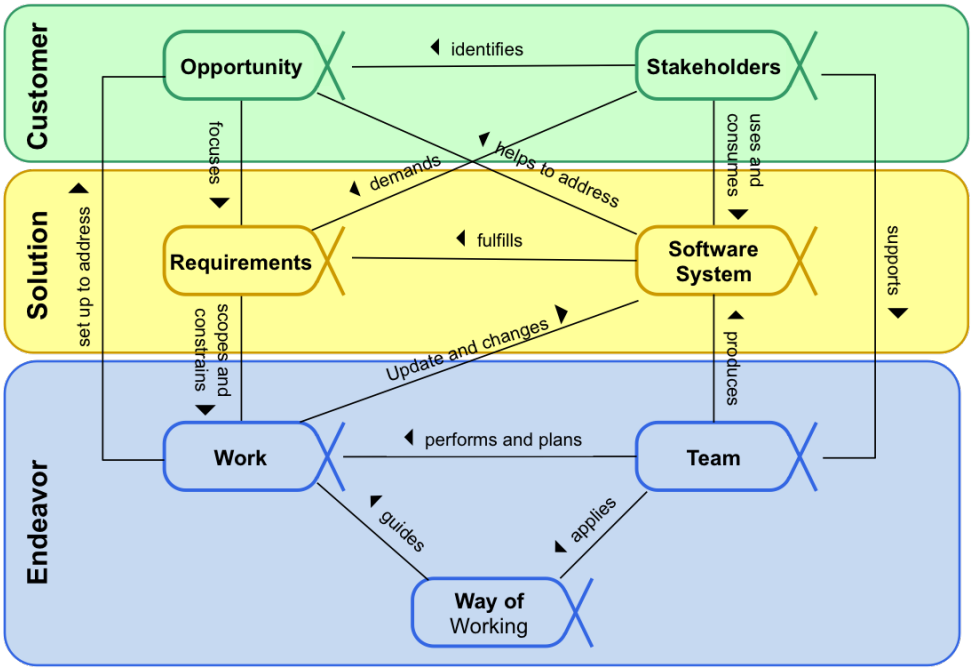
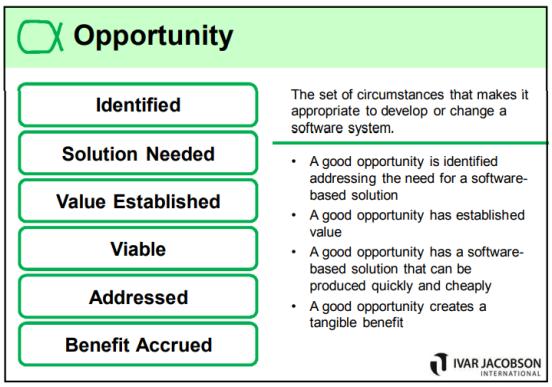
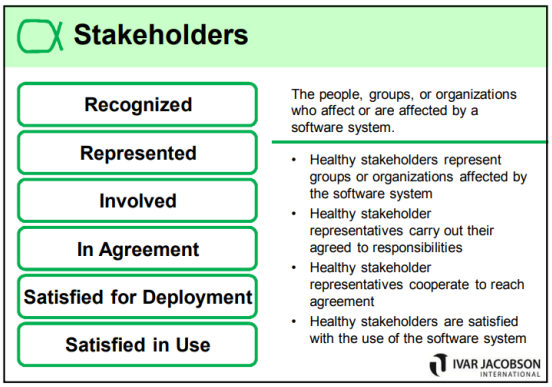


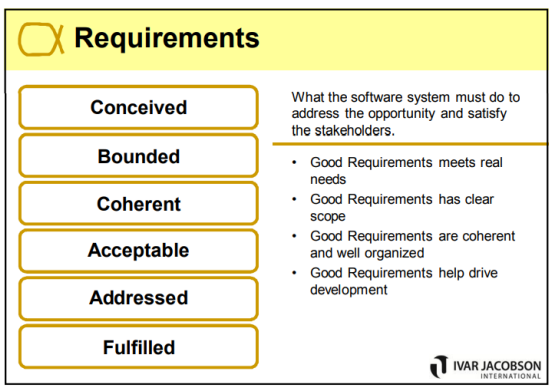
Figura 13: Elementi di un progetto.



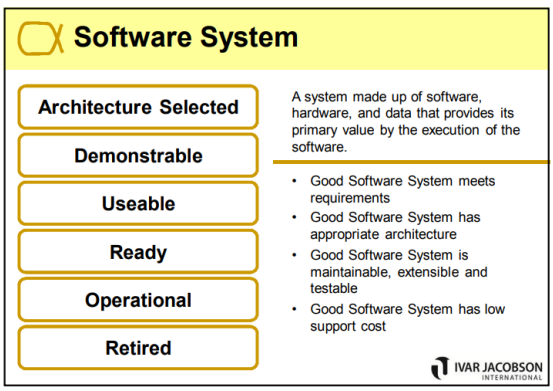
(a) Opportunity.



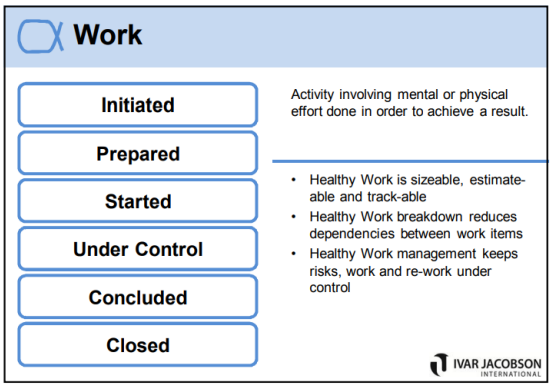
(b) Stakeholders.



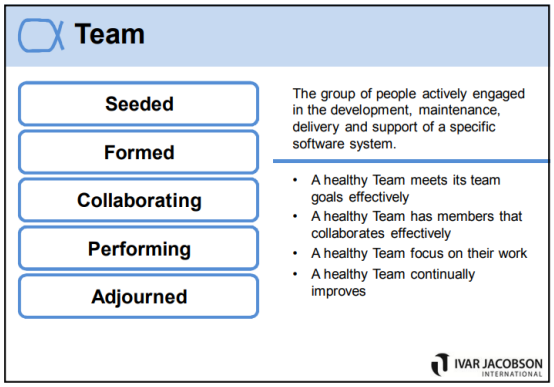
(c) Requirements.



(d) Software System.



(e) Work.



(f) Team.

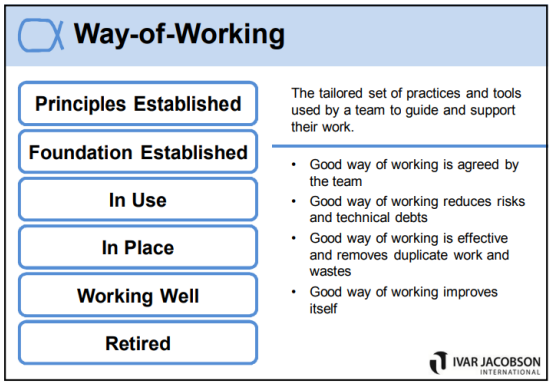


Figura 14: Way of working.

Molto importante all'interno di un progetto sono la Gestione di progettoe le "tre P" (PPP).

PROGRAMMATORE

È uno dei ruoli in un progetto. Sono la categoria più popolosa. Partecipano alla realizzazione e manutenzione del prodotto e hanno competenze tecniche, visione e responsabilità circoscritte.

PROGRAMMI VERIFICABILI

Dato che serve dotarsi di uno standard di codifica coerente con le esigenze di verifica, per scrivere programmi verificabili, serve regolamentare l'uso del linguaggio di programmazione tramite principi da riflettere nelle Norme di Progetto. In particolare per:

- Assicurare comportamento predicibile
- Usare criteri di programmazione ben fondati
- Ragioni pragmatiche

La prima tecnica per essere sicuri che non ci stiamo perdendo niente è il tracciamento. Gli obblighi di analisi statica del codice sono:

- **Analisi del flusso di controllo**: il modo in cui avanza il programma. Alcune istruzioni cambiano il flusso (ad es: go to, cicli, ecc) per cui è necessario usare la ricorsione con grande parsimonia
- **Analisi di flusso dei dati**: vedere l'ordine delle variabili. Si accerta che il programma non acceda a variabili prive di valore. Questa analisi è resa facile tramite incapsulamento e non si fa eseguendo il codice, ma prima.
- **Analisi del flusso d'informazione**: è il modo in cui valorizzo il dato. Determina la relazione tra ingressi e uscite di un'unità di codice. Bisogna qui capire se il valore informativo del dato è perso o meno.
- **Esecuzione simbolica**: dopo analisi di flusso di dati e d'informazione, so ora calcolare quale funzione dell'input produce l'output. L'uscita che vedo è la conseguenza di un flusso.
- **Verifica formale del codice**: usare le pre-condizioni, le post-condizioni e gli invarianti. È importante per vedere se il programma è corretto *by construction* e provare quindi la correttezza del codice sorgente rispetto alla specifica algebrica dei requisiti.
- **Analisi di limite**: limiti importanti sono *overflow* e *underflow*. Rispetto dei limiti (fare *range checking*).
- **Analisi d'uso di stack**: suddiviso in due:
  - Prima ragione di crescita: quando annido chiamate lo stack cresce. Lo stack è deciso all'elaborazione (quella è quindi la massima estensione).
  - Seconda ragione di crescita: i parametri che ci metto. Quando lo stack cresce oltre il suo confine cammina sull'heap, fare quindi attenzione che non tracimi (vedere le variabili del main dove stanno, se heap o stack)
- **Analisi temporale**: studiare le dipendenze temporali (latenza, tempestività) tra le uscite del programma e i suoi ingressi per verificare che il valore giusto sia prodotto al momento giusto
- **Analisi d'interferenza**: mostrare l'assenza di effetti d'interferenza tra parti isolate del sistema. Veicoli tipici di interferenza: memoria condivisa e I/O
- **Analisi di codice oggetto**: assicurare che il codice oggetto da eseguire sia una traduzione corretta del codice e che non sia stata introdotta nessuna omissione dal compilatore

PROOF OF CONCEPT

Poc: prototipo che mi dà una risposta alle domande sulla fattibilità della tecnologia selezionata. Prima di tutto capisco le domande (cosa ci dovrei fare) e poi elaboro le risposte. Dimostro poi che il lavoro si può fare e lo rendo accessibile al committente.
Rappresenta una baseline.

PROTOTIPO

Dà l'idea di ciò che sarà il prodotto. Serve per provare e scegliere possibili soluzioni e può essere usato in modo incrementale come la baseline, o "usa e getta", quindi iterativo.

PROVA

Una prove eseguibile è una procedura applicata a una batteria di prove.
Ciascun insieme di dati di ingresso, campioni delle classi di equivalenza, che producono un dato comportamento funzionale, costituisce un singolo caso di prova.

Q

QUALIFICA

Verifica + validazione.

QUALITÀ

L'insieme delle proprietà e delle caratteristiche di un prodotto che conferiscono ad esso la capacità di soddisfare esigenze espresse o implicite del cliente. Si vede chi fa (sulla base di qualche riferimento), chi usa (almeno il minimo obbligatorio) e chi valuta (etichettano). La qualità è *di prodotto* e *di processo*.
Per quel che riguarda la *qualità di prodotto* esistono:

- Sistema di Qualità: è la struttura organizzativa e le risorse messe in atto per perseguire la qualità, in cui c'è la supervisione di miglioramento continuo
- Piano della Qualità
- Controllo di qualità

La qualità è un poliedro: ha tante facce ed è in stretta correlazione a efficacia e valutazione. Riguardo alla *qualità di processo* bisogna definire il processo per controllarlo e migliorarne efficacia, efficienza ed esperienza. Usare inoltre buone metriche e strumenti di valutazione. Ci riferiamo qui ora allo standard ISO 9000.
Esiste inoltre un Manuale della qualità e degli strumenti di valutazione tra cui:

- **SPY**: *SW Process Assessment and Improvement* dà una valutazione oggettiva dei processi di una organizzazione per darne un giudizio di maturità e individuare azioni migliorative
- **CMMI**
- **ISO 15504**

R

READY

Vuol dire che nessuna parte manca e la documentazione è pronta. Gli stakeholders hanno accettato il prodotto e vogliono che diventi operativo.
È l'ultimo degli stati di progresso del SEMAT per la progettazione (vedi immagine).

REALIZZAZIONE

Attua ciò che è stato progettato durante la progettazione software: avviene quindi la stesura del codice SENZA INVENTARE. Infine avviene l'attività di collaudo che è il nostro passo di uscita sapendo l'esito che avrà.

REQUIREMENTS

Idea di soluzione per soddisfare il bisogno (quindi astratta). È la descrizione documentata di una *capacità*

- Necessaria a un utente per raggiungere un obiettivo (vista dal lato del cliente)
- Che un sistema deve possedere per adempiere a un obbligo (vista dal lato dello sviluppatore)

I requisiti devono essere tutti identificabili e verificabili. Vengono classificati, per facilitare la comprensione, la manutenzione e il tracciamento, in base ad

- **Attributi di prodotto**: rispondono a *Cosa devo fare?* (requisiti funzionali, prestazionali e di qualità)
- **Attributi di processo**: rispondono a *Come devo farlo?* (requisiti di vincolo)

Ogni tipo di requisito può essere diretto o indiretto, implicito o esplicito ma ognuno deve essere verificabile tramite:

- **Per requisiti funzionali**: test, dimostrazione frontale, revisione
- **Per requisiti prestazionali**: misurazioni
- **Per requisiti qualitativi**: tecniche ad hoc
- **Per requisiti dichiarativi (vincoli)**: revisione

In base all'utilità strategica possono inoltre essere classificati in:

- **Obbligatori**: irrinunciabili secondo gli stakeholders
- **Desiderabili**: non strettamente necessari ma danno valore aggiunto
- **Opzionali**: relativamente utili

Questo perché devo capire quali requisiti sono irrinunciabili. I requisiti sono quindi elastici e dipendono da me, da quanto sono veloce, ecc.

Qualità che ci devono essere secondo IEEE 830:

- **Non ambiguo** ovvero il requisito deve essere scritto in maniera concisa per non creare confusione (ma attenzione perché non c'è una metrica)
- **Corretto**
- **Completo** perché c'è tutto ciò che ci deve essere
- **Verificabile**
- **Consistente** perché non possono essere contraddittori
- **Modificabile** con la gestione dei cambiamenti, ma l'indice che ogni requisito ha non lo do io (altrimenti se modifico qualcosa devo riordinare tutto) e deve essere significativo
- **Tracciabile**
- **Ordinato per prevalenza**

Durante l'analisi dei requisiti avviene inoltre la verifica dei requisiti:

- Viene eseguita su un documento organizzato tramite walkthrough o ispezione (lettura mirata)
- Si ricerca chiarezza espressiva, che non è il linguaggio naturale
- Si ricerca chiarezza strutturale, ovvero si controlla la separazione tra requisiti funzionali e non-funzionali e che la classificazione dei requisiti sia precisa, uniforme e accurata
- Si ricerca atomicità e aggregazione, quindi i requisiti elementari e le correlazioni tra di essi chiare ed esplicite

E anche la gestione dei requisiti.

RESPONSABILE

È uno dei ruoli in un progetto. Partecipa al progetto per tutta la sua durata rappresentando il progetto presso il fornitore e il committente. Accentra le responsabilità di scelta e approvazione, quindi deve avere conoscenze e capacità tecniche per valutare (rischi e scelte varie). Ha molte responsabilità, nello specifico su: pianificazione, gestione delle risorse umane, controllo, coordinamento e relazioni con l'esterno.

REVISIONE DEI REQUISITI

Revisione 1.
Ha la funzione di concordare con il cliente una visione condivisa del prodotto atteso. Prevede:

- Analisi dei Requisiti
- Piano di Progetto
- Piano di Qualifica

- Norme di Progetto
- Studio di Fattibilità

Fa l'Audit Process.

REVISIONE DI ACCETTAZIONE

Revisione 4.
Ha la funzione di collaudare il sistema per accettazione da parte del committente e accertarsi del soddisfacimento di tutti i requisiti utente fissati nella RR. Prevede:

- Piano di Qualifica (quarta versione)
- Piano di Progetto(quarta versione) con consuntivo finale
- Manuale Utente e Manuale Sviluppatore (entrambe seconda versione)

Fa l'Audit Process.

REVISIONE DI PROGETTAZIONE

Revisione 2.
Ha la funzione di accertare la realizzabilità. Prevede:

- Technology Baseline
- Piano di Qualifica (seconda versione)
- Piano di Progetto (seconda versione)
- Norme di Progetto (seconda versione)

Fa il Joint Review Process.

REVISIONE DI QUALIFICA

Revisione 3.
Ha la funzione di approvare l'esito finale delle verifiche e attivare la validazione. Prevede:

- Product Baseline
- Piano di Qualifica (terza versione)
- Piano di Progetto (terza versione)
- Manuale Utente e Manuale Sviluppatore

Fa il Joint Review Process.

RIUSO

Il riuso può essere:

- **Occasionale**: perchè "copia e incolla" opportunistico, che ha quindi un basso costo, ma scarso impatto
- **Sistematico**: perchè è un "copia e incolla" intelligente, che ha un maggior costo ma maggior impatto

Nel breve periodo il riuso diventa più che altro un costo.

RUOLI

Il ruolo è una persona che si occupa di una *funzione aziendale* assegnata in un progetto. Tra queste funzioni aziendali troviamo:

- **Sviluppo**: quindi la persona dovrà avere responsabilità tecnica e realizzativa
- **Direzione**: quindi la persona dovrà avere responsabilità decisionale.
- **Amministrazione**: ovvero gestione del supporto ai processi.
- **Qualità**: ovvero gestione della ricerca di economicità.

A seguire, l'elenco dei ruoli all'interno della gestione del progetto (ricordiamo che non vogliamo sovrapposizioni):

- **Analista**: fa l'analisi dei requisiti. Capisce ciò di cui c'è bisogno nel mio prodotto, per cui è importante che mettersi nei panni dell'utente.
- **Progettista**: pensa ad una possibile soluzione con qualità desiderabili (**NB**: non implementa).
- **Programmatore**: concretizza la soluzione che il progettista ha pensato (**NB**: non inventa).
- **Verificatore**: dice se il lavoro del programmatore è conforme alle attese. Deve poi relazionarsi ed essere di supporto.
- **Responsabile**: ha "onere e onore", in particolare ha tanti oneri. Coordina il team e garantisce che non ci siano intoppi o rallentamenti.
- **Amministratore**: garantisce che funzioni bene il nostro sistema informatico (che sia scalabile, ecc).

S

SCALABILITÀ

È la caratteristica di un sistema software o hardware facilmente modificabile nel caso di variazioni notevoli della mole o della tipologia dei dati trattati.
Garantisce prestazioni.

SEMAT

Software Engineering Method and Theory fornisce teorie, principi provati e best practices per l'Ingegneria del Software.

SISTEMA DI QUALITÀ

Riferimenti **oggettivi** che ci dicono come stiamo lavorando.

SLACK

Significato letterale:"lasco". Con questo si intende il margine che posso consumare (per esempio, nel caso di un ritardo di attività che devo finire di completare).

SOFTWARE DETERMINISTICO

In ogni stato del prodotto SW in cui mi trovo, so che posso avere solo "un'unica e sola uscita". Devo sempre sapere lo stato in cui sono, quindi avere ben chiaro quello precedente e quello seguente.

SOFTWARE ENGINEERING

L'Ingegneria del Software è una disciplina volta a realizzare prodotti software talmente impegnativi da richiedere lo svolgimento di attività collaborative. Agisce garantendo efficacia ed efficienza durante tutto il ciclo di vita del prodotto.
È importante sottolineare che l'Ingegneria del Software non ha a che vedere solo con l'informatica, ma anche con alcune aree della matematica discreta, ricerca, statistica, psicologia ed economia. L'Ingegneria del Software inoltre adotta un determinato approccio.

SOFTWARE SYSTEM

Soluzione realizzabile (concretizza requirements).

SOLUTION

Secondo elemento di un progetto secondo SEMAT. Esso comprende requirements e software system.

STAKEHOLDER

Traduzione di *portatori di interesse*, sono le persone influenti per il prodotto: dicono se una certa opportunità è buona. Possono essere chi usa il prodotto (cliente), chi compra il prodotto (committente), chi sostiene i costi di realizzazione (fornitore), chi verifica l'attuazione di processi (eventuali regolatori).
È uno degli elementi di un progetto (vedi immagine).

STANDARD DI PROCESSO

Nascono per iniziativa del committente al fine di facilitare controllo, collaudo e accettazione. Esistono standard come:

- **Modello di azione**: definiscono procedure o processi
- **Modello di valutazione**: sono modelli più generali e identificano la best practice

STANDARD IEEE 830-1998

Questo standard delinea la struttura che deve avere il documento Analisi dei Requisiti e le qualità che devono possedere i requisiti. *Recommended Practice for Software Requirements Specifications* dice che la specifica deve essere:

- Priva di ambiguità
- Corretta
- Completa
- Verificabile
- Consistente
- Modificabile
- Tracciabile
- Ordinata per rilevanza

Per la restante documentazione relativa alla struttura dell'AdR, vedere il link qui.

STANDARD ISO 12207

È un *modello di azione* ed il modello più noto. Contiene tutti i processi significativi ad alto livello:

- Identifica i processi di ciclo di vita del software
- Ha struttura modulare
- Specifica le responsabilità sui processi
- Specifica i prodotti dei processi

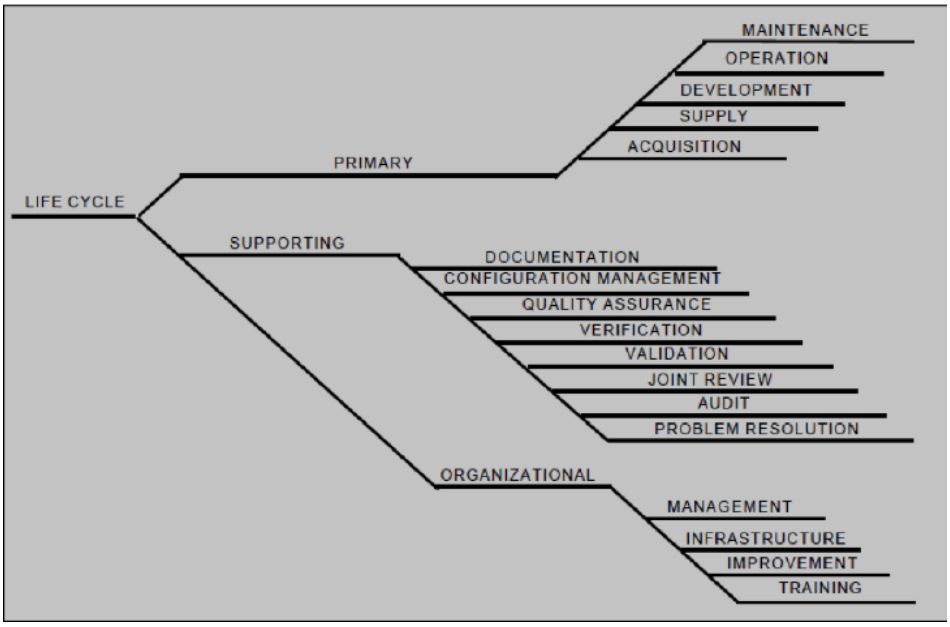


Figura 15: Primary, supporting and organisational life-cycle processes.

Almeno un *processo primario* deve sempre esistere:

- **Acquisizione**: gestione dei sotto-fornitori
- **Fornitura**: gestione dei rapporti con il cliente
- **Sviluppo**: non è sempre detto che abbia rapporti con il cliente
- **Gestione operativa/utilizzo**: installazione ed erogazione dei prodotti
- **Manutenzione**

Processi di supporto:

- **Documentazione**
- **Accertamento della qualità**
- **Gestione delle versioni e delle configurazioni**
- **Verifica**
- **Validazione**
- **Revisioni congiunte con il cliente**
- **Verifiche ispettive interne**
- **Risoluzione dei problemi**: con gestione dei problemi

Trasversali rispetto ai singoli progetti sono invece i *processi organizzativi*:

- **Gestione dei processi**
- **Gestione delle infrastrutture**
- **Miglioramento del processo**
- **Formazione del personale**

STANDARD ISO 14598

ISO/IEC 14598:1999 fornisce il *modello di valutazione* che non cambia nel tempo e col contesto. La metrica che adotta è un modo per dare un significato condiviso e non ambiguo a delle caratteristiche.

STANDARD ISO 15504

Software Process Improvement Capability dEtermination
Standard di qualità nato per armonizzare 12207 e 9001. È un *modello di valutazione* e si occupa di:

- **Identificazione degli stakeholder**: destinatari dei risultati, responsabili dei processi valutati
- **Scelta tra valutazione e miglioramento**: risultato a uso esterno o interno, approccio formale (*audit*) o meno (*self-assessment*)
- **Definizione della portata**: processi inclusi nella valutazione e indicatori di valutazione

STANDARD ISO 15939

(Sappiamo solo che esiste ma non è da usare perchè è costoso)
Misura e ha forma di ciclo, quindi non ha fine. Il mio cruscotto cambia in base al contesto, non è scelto a priori e basta, se no diventa vecchio (esempio: quando faccio codice guardo la qualità per il codice, quando faccio requisiti ho un altro cruscotto).

STANDARD ISO 25000

ISO/IEC 25000:2014 è un insieme dello standard 9126 e 14598. Si riferisce al software, *SQuaRE*: *Software product Quality Requirements and Evaluation.*

STANDARD ISO 9000

La famiglia delle norme ISO 9000 tratta i fondamenti dei modelli di qualità, neutri rispetto al dominio di applicazione.

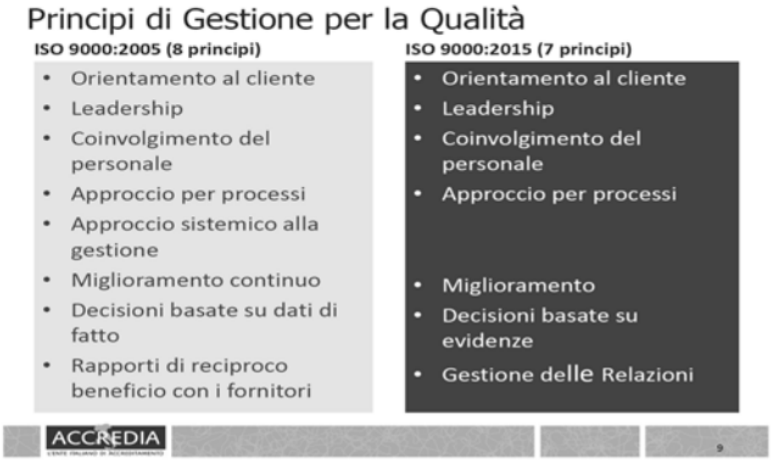


Figura 16: I principi ISO 9000.

STANDARD ISO 9001

Applica lo standard 9000 ai sistemi produttivi. È la certificazione per la valutazione dei fornitori di prodotti o servizi.
Per il miglioramento dei risultati è stato in seguito creato il 9004.

STANDARD ISO 9126

Standard per la qualità del prodotto. Fatto di due parti:

- Come si definiscono le caratteristiche rilevanti
- Come si misurano (con quali metriche) per poterle valutare

Tre diverse visioni tutte da considerare:

- **Visione esterna**: ciò che si osserva del prodotto relativo all'esecuzione
- **Visione interna**: come è fatto rispetto a cosa fa
- **Visione in uso**: come vedo il prodotto quando devo usarlo

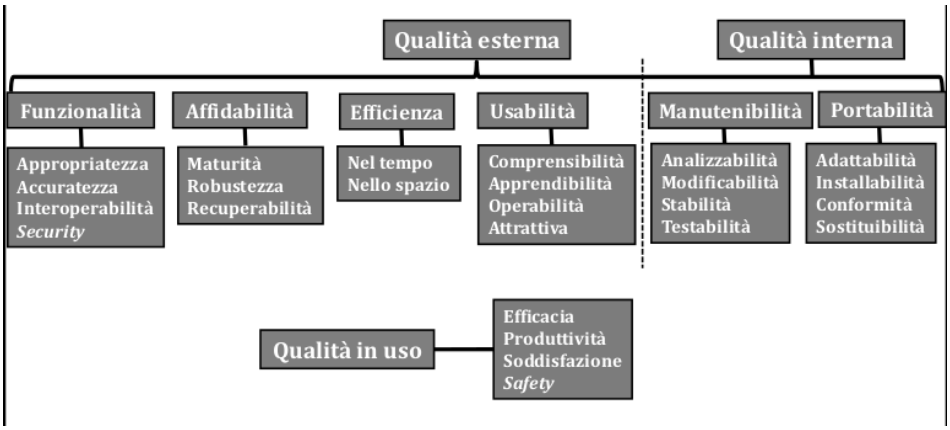


Figura 17: Le diverse visioni delle qualità di ISO 9126:2001.

STATEMENT COVERAGE

Strumento che si ricorda quali *statement* il test ha attraversato. Si ha quindi copertura al 100% quando i test effettuati sull'unità sono sufficienti a eseguire almeno una volta tutti i comandi dell'unità con esito corretto.
(Un esempio di differenza con branch coverage su StackOverflow).

STATO

È il valore assunto da variabili di stato, conseguente ad azioni precedenti su di esse. Viene deciso dal *Software Engineer*.

STUB

Sono dei sostituti (come i mock), ovvero rappresentano ciò che è chiamato dal test, ma che ancora non c'è (per esempio evita ClassNotFound).

STUDIO DI FATTIBILITÀ

Non è un documento pubblico, ma deve rimanere agli atti come un ragionamento ben fondato. È quindi un prodotto interno e definisce il rapporto tra le parti.
Si capisce qui cosa si è capaci di fare, perché innanzitutto si valutano rischi, costi e benefici nell'ottica del cliente e del fornitore. Si valuta se procedere, con l'obiettivo di restare entro un costo massimo prefissato e con le conoscenze immediatamente disponibili o un piano di formazione. Si studiano:

- Gli strumenti e le tecnologie per la realizzazione
- Le soluzioni algoritmiche e architetturali
- Le piattaforme idonee all'esecuzione
- Il costo di produzione rispetto alla redditività

Inoltre le attività che vengono fatte sono:

- Individuazione dei rischi
- Valutazione delle scadenze temporali (con conseguente studio delle risorse disponibili rispetto a quelle necessarie)
- Valutazione delle possibili strategie alternative

T

TASK

È l'equivalente inglese di *attività*: complesso di azioni dirette alla realizzazione di un obiettivo.
È diverso da compito: parte di lavoro che si assegna ad altri o che qualcuno prefigge a se stesso di fare.

TECHNOLOGY BASELINE

Dimostrare, al committente e a noi stessi, di disporre delle tecnologie, librerie e framework necessari per lo sviluppo del prodotto, portando una baseline. Ne dimostriamo l'adeguatezza tramite Proof of Concept ed è soggetta a verifica Agile.
Fa parte del periodo di RP.

TEMPO/PERSONA

Quanto tempo produttivo sta usando quella persona (o profilo). La sommatoria mi dà il costo complessivo.

TEST

Prima di mettere insieme tutti i vari programmi in un prodotto SW, bisogna accertarsi che ogni piccola parte funzioni tramite test. Per questo conviene adottare la *Continuous Integration* che è una metodologia intelligente. Essa mi fa scegliere prima i vari test che mi andranno a testare una piccola parte di programma, poi quelli a cui al momento ne manca ancora un'altra (esempio: l'interfaccia senza database).
I principali tipi di test sono:

- Test di unità
- Test d'integrazione
- Test di sistema
- Test di regressione
- Test di accettazione (collaudo)

E devono essere:

- **Ripetibili**: per questo si specifica ambiente di esecuzione, attese e procedure
- **Automatizzati**: per questo si usano strumenti come *driver*, *stub* e *logger* (sono mock)
- **Oggettivi**: non personalizzati

Ma fare i test in modo esaustivo è impossibile.
N.B: tutte le regole dei test vengono decise nella progettazione.

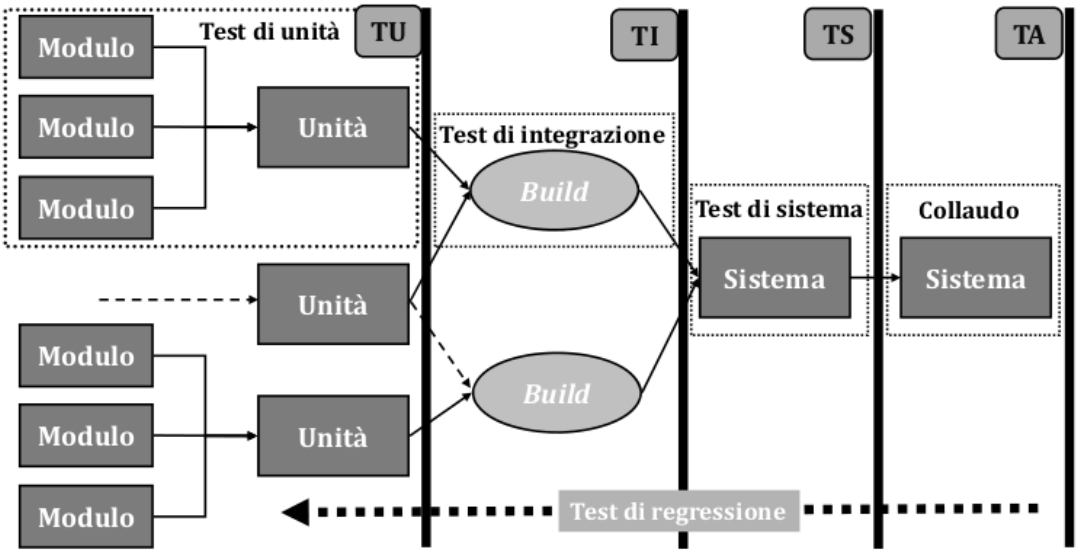


Figura 18: Tipi di test (analisi dinamica).

I test comportano l'esecuzione dell'oggetto di verifica e devono dare misure di qualità. Ogni test deve avere un obiettivo:

- Chiaro
- Utile
- Decidibile, perché c'è l'oracolo, ovvero deve produrre un esito verificato rispetto a un comportamento atteso

E il posto in cui inserisco queste cose è il piano di qualifica (rispettando il modello a V, ovvero prima definisco i test di validazione e sistema, poi quelli d'integrazione e infine i test di unità).

Il verificatore controlla se ci sono problemi nei test. Se ci sono, essi sono dovuti a:

1. Mistake
2. Fault
3. Error
4. Failure.

Più obiettivi ha un test e peggio è (perché diventa difficile realizzarlo, farne il tracciamento, ecc). Più semplice è il SW da scrivere, più semplice sarà il test.
La strategia richiede di trovare un bilanciamento tra la quantità minima e massima di casi di prova rispettando la legge del rendimento decrescente. È importante perché i test costano e questo non è banale.
Secondo Bertrand Meyer: *"bisogna smantellare quello che ha in pancia un fault"*, ovvero bisogna sempre cercare di far fallire un test.

I test si specificano (*Specifica*), poi vanno tradotti in (*Codifica*) un eseguibile (da Codifica in poi voglio automatizzazione), collegati all'oggetto di test (*Compilazione*), devo farli eseguire (*Esecuzione*) e registrare l'esito (*Analisi*). Infine l'output di un test si mette in un file chiamato file di log.

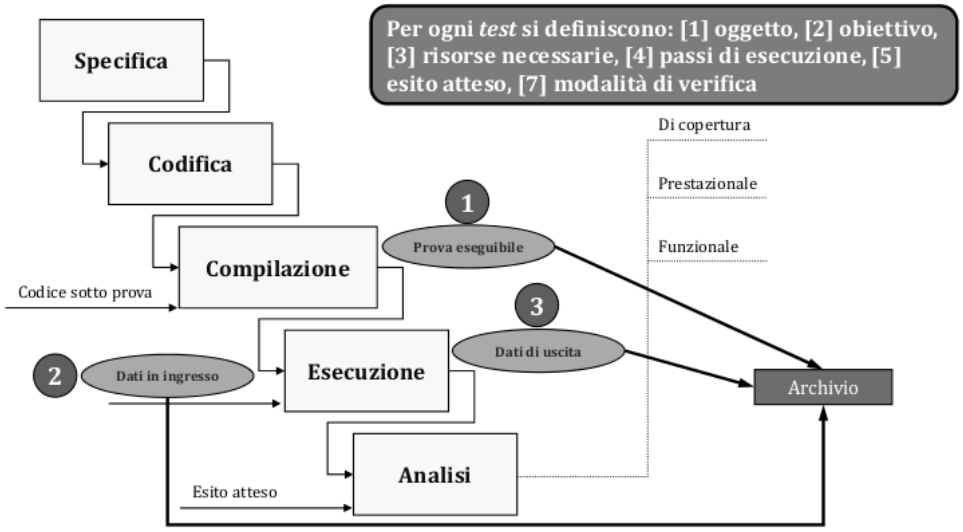


Figura 19: Attività di prova.

Si lavora per classi di equivalenza, in particolare 3:

- Valori non ammessi
- Valori barriera
- Valori ammessi entro barriere

Ogni insieme ha un rappresentate nei test. È facile ordinare gli interi qui dentro le classi di equivalenza, ma non altri valori come i numeri reali.
Esistono inoltre 2 categorie di test principali:

- **Funzionale**: *black-box* si occupa di vedere il tipo di uscita di quel test e credere che al suo interno vada tutto bene
- **Strutturale**: *white-box* si occupa di cosa fa il test e non dell'effetto che produce, ricercando la massima copertura

Puntiamo ad avere massima copertura tramite i due *coverage* (fattori di copertura):

- **Statement Coverage**
- **Branch Coverage**

È difficile avere proprio il 100%, quindi aver soddisfatto tutti i requisiti è tecnicamente impossibile, anche se è il nostro obiettivo.

TEST CASE

È una singola specifica di un singolo test. *Case* è un "insieme di" e, più specificatamente, un insieme di parametri:

- Ingresso
- Uscita
- Oggetto di prova
- Ambiente di sviluppo

TEST DI INTEGRAZIONE

Verifica il residuo, che non possiamo accettare da una componente singola, mettendo insieme due unità con una *build*. Sono idealmente quindi parti indipendenti messe insieme per collaborare, coese. L'integrazione, se c'è una buona architettura, è anche parallelizzabile.
La *logica* di integrazione funzionale:

1. Seleziona le funzionalità da integrare
2. Identifica le componenti che svolgono quelle funzionalità
3. Ordina le componenti per numero di dipendenze crescenti
4. Esegue l'integrazione in quel determinato ordine.

Più test di questo tipo faccio, più piccola è la parte che sto testando.
Il test di integrazione rileva *problemi* quali:

- Errori residui nella realizzazione dei componenti
- Modifiche delle interfacce o cambiamenti nei requisiti
- Riuso di componenti dal comportamento oscuro o inadatto
- Integrazione con altre applicazioni non ben conosciute

La strategia si basa sostanzialmente di proseguire per passi, aggiungendo "pezzi" fino al completamento. Adotta quindi una strategia incrementale. Assemblo prima i produttori e poi i consumatori, seguendo la mia architettura. Dovrebbe essere inoltre reversibile perché se, per esempio, assemblo A e B e questo non va bene, allora devo avere la possibilità di tornare indietro.
Secondo gli approcci top-down e bottom-up qui, più in alto ci sta l'interazione con l'utente, più in basso ci sta la persistenza.
Scelgo **bottom-up** se l'utente non ha ancora un punto da cui iniziare a lavorare. Si integrano prima le parti con maggiore utilità e minore dipendenza. Questa strategia riduce il numero di stub necessari al test, ma ritarda la disponibilità di funzionalità di alto livello.
Scelgo **top-down** se è utile negoziare con l'utente. Questa strategia comporta l'uso di molti stub, ma integra a partire dalle funzionalità di più alto livello. Il test di integrazione ha tanti test quanti ne servono per accertarsi che tutti i dati scambiati attraverso ciascuna interfaccia siano conformi alla loro specifica e accertarsi che tutti i flussi di controllo previsti siano stati effettivamente provati.

TEST DI REGRESSIONE

Collegato fortemente agli altri test. È una ripetizione selettiva di test di unità, d'integrazione e di sistema. Si assicura che una modifica fatta su un'unità per correggere un problema, non faccia danno ad altre causando regressione. La regressione si riduce se il sistema è scarsamente accoppiato (per esempio tramite incapsulamento ecc).
Torno indietro nel mio codice per correggere se qualcosa è andato male. Ma anche questo correggere può produrre errori. Con questo test dimostro che non torno indietro.

TEST DI SISTEMA

Se le componenti del programma sono buone, faccio il sistema. Questo tipo di test è per il fornitore e serve per dire che è tutto a posto controllando che tutti i requisiti siano soddisfatti (quindi per dimostrare la conformità del prodotto).
I test di sistema hanno inizio quando è completato il test d'integrazione e sono per definizione a scatola chiusa, nera. È una *self-fullfilling-prophecy*: non ho ansia, sono sicuro. Il test di sistema viene fatto quando faccio i requisiti (e ciò mi fa capire che non ci piacciono requisiti grandi e confusionari, ma sminuzzati).

TEST DI UNITÀ

Problema che non so individuare l'unità. L'unità si sceglie quando si progetta. La *liability* di ogni persona vogliamo che sia piccola in modo che l'unità sia facilmente verificabile e ragionevole. Se un'unità quindi non è determinata da un linguaggio di programmazione, sarà un aggregato di cose che noi chiamiamo moduli.
Il test di unità segna il primo *gate* e si svolge con il massimo grado di parallelismo.

TEST SUITE

Suite è un completo, per cui questo è un insieme di test case, ma completo.

TOP-DOWN

Si tratta di un apporcio che ci fa proseguire dal tutto alle parti. Fondamentalmente è un'esplorazione funzionale, senza preconcetti.

TRACCIAMENTO

Tracciamento dei requisiti:
Si occupa della gestione dell'evoluzione dei requisiti, motivo per cui è essenziale sapere sempre a che punto si è nella copertura (= soddisfacimento) dei requisiti. Il tracciamento dei requisiti è essenziale per il controllo sistematico di conformità. Mi dà le risposte alle domande: "" *Ho tutto o no? Ho cose superflue?*". Ogni parte del documento dell'Analisi dei Requisiti c'è per bisogni impliciti o espliciti (tipo dal capitolato d'assalto), quindi il tracciamento tiene conto di tutti i collegamenti (anche fonti interne come i verbali, ecc).

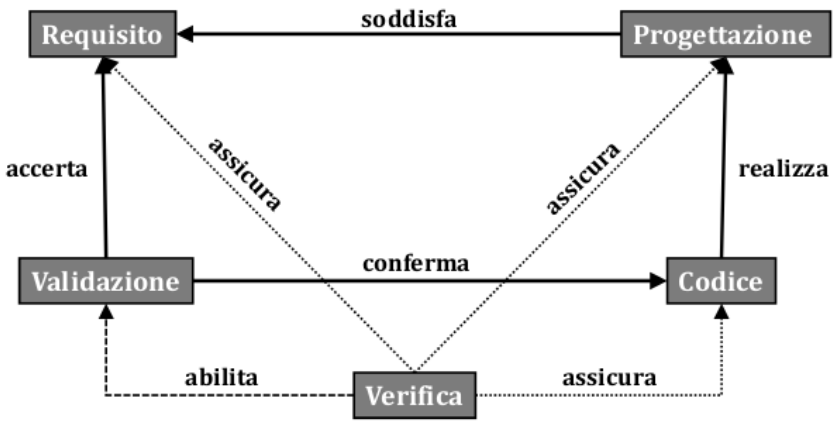


Figura 20: Visualizzazione ordine del tracciamento dei requisiti.

Tracciamento sul codice:
Il tracciamento è qui vitale perché devo essere sicura di quello che sto facendo e di non fare cose oziose. Dimostra completezza ed economicità della soluzione e connette sempre parti correlate, infatti ha luoghi su ogni passaggio dello sviluppo (ramo discendente della figura a V) e su ogni passaggio della verifica (ramo ascendente). Può essere altamente automatizzato. Particolari stili di programmazione facilitano il tracciamento, per esempio: assegnare singoli requisiti elementari a singoli moduli del programma richiede una sola procedura di prova e questo ne semplifica anche la verifica.

U

UNITÀ

Dividere l'organizzazione del lavoro di programmazione fino al punto in cui non conviene più, ovvero in cui si giunge all'unità. Essa corrisponde ad una funzionalità e può essere per esempio una classe o dei dati.
Ciò che ho prodotto si chiamano moduli. Un'unità infatti può essere anche un insieme di più moduli.
La specifica di ogni unità architetturale deve essere documentata.

USABLE

Milestone usable, ovvero il sistema è utilizzabile, ha le caratteristiche desiderate, e infatti può essere operato dagli utenti. Le funzionalità e le prestazioni richieste sono state verificate e validate e la quantità di difetti è accettabile.
Si rivolge alla Product Baseline ed è uno degli stati di progresso del SEMAT per la progettazione.

V

VALIDARE

"Did I build the right system?".
Ha come obiettivo l'accertarsi che il prodotto sia conforme alle attese, infatti si fa sull'esito di sviluppo. Prevede test di sistema e collaudo.

VALUTAZIONE

Determinazione del valore da assegnare a cose o fatti ai fini di un giudizio. Varia in base ai destinatari perché hanno diverse aspettative.
Notare che la qualità fa da appoggio per la valutazione.

VERIFICARE

"Did I build the system right?".
Accertare che l'esecuzione delle attività di processi svolti nella fase in esame non abbia introdotto errori nel prodotto, infatti va fatta rivolta ai processi.
Esistono diverse forme di verifica:

- **Dinamica**: richiede l'esecuzione del programma e si fa tramite test
- **Statica**: non richiede l'esecuzione del prodotto. La più semplice e umana è la lettura. Ci sono due modi di lettura:
  - Waltkthrough
  - Inspection: controllo mirato a cose chiamate *lista di controllo*

La verifica serve per scovare i problemi e risolverli tempestivamente tramite *problem solution*.

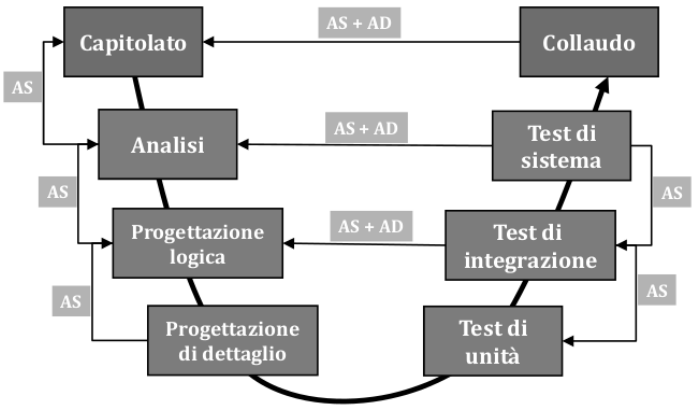


Figura 21: Verifica e validazione nello sviluppo.

La programmazione non deve costituire ostacolo alla verifica, anche se spesso lo fa. L'obiettivo è quindi facilitare la verifica, sebbene esista anche il conflitto tra essere economici (poco) e sapere cosa accade (tanto).
Morale: serve uno standard di verifica che si protrae in lungo, rendendo agevole la manutenzione, e che eviti assolutamente la verifica retrospettiva (ovvero, ho fatto e ora rivedo

cos'ho fatto sistemando quello che ho sbagliato). Questo perchè il costo di rilevazione e correzione di errori cresce con l'avanzare dello sviluppo.

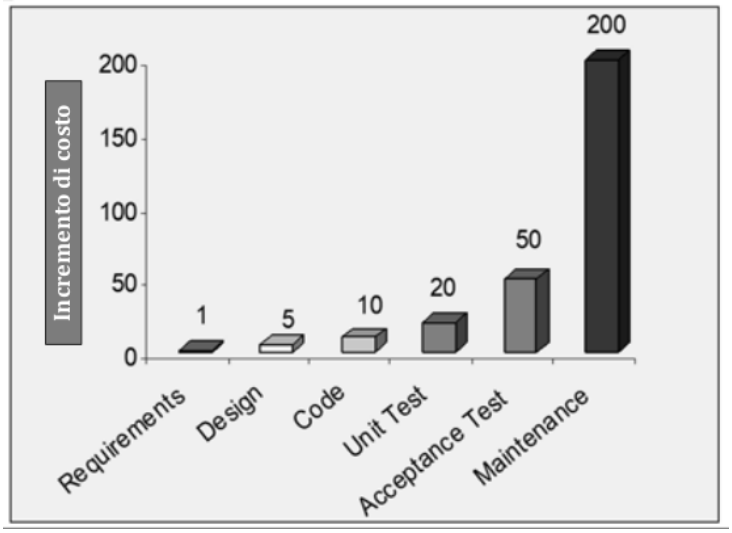


Figura 22: Costo di correzione di errori.

Un buon approccio è quindi verificare tramite correttezza per costruzione. È inoltre importante e di nostro interesse scrivere programmi verificabili.
NB: la verifica si può parallelizzare.

VERIFICATORE

È uno dei ruoli in un progetto. Sono presenti per l'intera durata del progetto. Hanno capacità di giudizio e di relazione, oltre ad avere competenze tecniche, esperienza professionale e conoscenza delle norme.

VERSIONE

È un'istanza di un determinato *configuration item*. Una versione è un'istanza di un configuration item diversa dalla precedente.

W

WALKTHROUGH

Tecnica che consiste nel cercare la presenza di difetti in ogni dove, non sapendo con esattezza dove cercare. È un metodo di lettura pratico come Inspection, la cui efficacia dipende dall'esperienza dei verificatori. La strategia, per il codice, prevede il percorrerlo tutto simulandone possibili esecuzioni.
Le sue attività sono divise in 4:

1. Pianificazione
2. Lettura
3. Discussione
4. Correzione dei difetti

E in ognuna svolta deve essere fatta la documentazione.

WORK BREAKDOWN STRUCTURE

Equivalente inglese di "struttura di scomposizione del lavoro" è una struttura gerarchica di attività che si compongono di sotto-attività non necessariamente sequenziali e univocamente identificate. Tratta quindi la scomposizione di un progetto in componenti più piccole. Questa suddivisione è particolarmente di aiuto al project manager nell'organizzazione delle attività di cui è responsabile.
È uno strumento per la pianificazione.

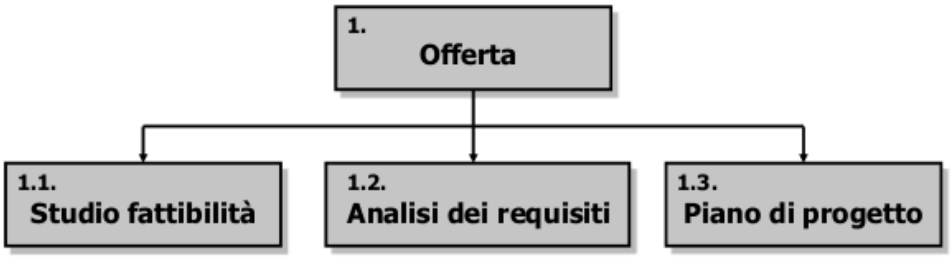


Figura 23: Esempio di diagramma di WBS.

Z

ZERO-LATENCY

Si tratta di *fare tutto subito*, appena l'impegno mi è stato assegnato.

ZERO-LAXITY

Si tratta di *fare tutto all'ultimo*, ovvero tenersi zero margine.