



INGEGNERIA DEL SOFTWARE

Appunti Lezioni – Anno 2020/2021



GAIA SERAVELLI

TIPOLOGIE Software

- SU COMMESSA DEL CLIENTE
- PACCHETTO
- COMPONENTE
- SERVIZIO

FORME DI Manutenzione

CORRETTIVA

corregge i difetti
trovati

ADATTIVA

adatta il sistema
alle variazioni dei
requisiti
20% DEI COSTI

EFFETTIVA o
EVOLUTIVA
aggiunge
funzionalità
60% DEI COSTI

Qualità interne

RIUSABILITÀ

PORTABILITÀ

VERIFICABILITÀ
INTEROPERABILITÀ
RIPARABILITÀ + EVOLVIBILITÀ

=
MANUTENIBILITÀ

COMPRENSIBILITÀ

Qualità esterne

CORRETTEZZA
AFFIDABILITÀ
ROBUSTEZZA
PRESTAZIONI
USABILITÀ

PRODUTTIVITÀ
TEMPESTIVITÀ
VISIBILITÀ

MISURARE IL SW

METRICHE RICONOSCIUTE:

- LINES OF CODE (LOC)
 - FUNCTION POINTS
 - COMPLESSITÀ CICLOMATICA
 - MEAN TIME TO FAILURE (ogni quanto si blocca)
 - MEAN TIME BETWEEN FAILURE
- DIMENSIONE
- COMPLESSITÀ
- AFFIDABILITÀ

La misurazione fornisce un numero
astratto x che dobbiamo trasformare in
costo gg/uomo.

INGEGNERIA DEL SW

PROGRAMMI,
DOCUMENTAZIONE,
LIBRERIE, DATI DI
CONFIGURAZIONE...

APPROCCIO SISTEMATICO, DISCIPLINATO e
QUANTIFICABILE allo SVILUPPO e MANUTENZIONE del sw.

ATTIVITÀ:

1. PIANIFICAZIONE
2. ANALISI DEI REQUISITI (cosa fare?)
3. PROGETTAZIONE (come farlo?)
4. REALIZZAZIONE con MAX. EFFICIENZA ed EFFICACIA
5. VERIFICA e VALIDAZIONE
6. MANUTENZIONE

ANALISTA DI SISTEMA

- ricava i requisiti
- interagisce col CLIENTE
- comprende l'area applicativa

ATTORI

- ANALISTA DI PRESTAZIONI
- analizza le prestazioni del sistema.

Sviluppatore Senior

- coordina lo sviluppo.

AREE APPLICATIVE

1. DATABASE:



CONFIDENZIALITÀ
dei DATI

INTEGRITÀ
dei DATI

DISPONIBILITÀ
dei DATI

USABILITÀ

PRESTAZIONI
delle
TRANSAZIONI

2. SISTEMI IN TEMPO REALE:



devo rispondere a certi eventi entro
un tempo prefissato e limitato.
Sono ORIENTATI AL CONTROLLO e si base=mo su un PIANIFICATORE.

TEMPO DI RISPOSTA AFFIDABILITÀ SAFETY

3. SISTEMI DISTRIBUITI:



LIVELLO DI DISTRIBUZIONE

TOLLERANZA CADUTA DI 1 o più PC

TOLLERANZA del PARTIZIONAMENTO

4. SISTEMI EMBEDDED:



Sono sistemi nei quali il sw è solo
uno dei componenti e spesso non ha
interfaccie rivolte all'utente finale, ma
solo verso altri componenti del sistema
che esso controlla.

Per stimare i costi di un SW si tengono in considerazione i COCOMO : CONSTRUCTIVE COST MODEL

LOC

Nom misura costi, tempi e rischi.

Misura la DIENSIONE del SW e lo SFORZO DI IMPLEMENTAZIONE.

Dipende molto dal LINGUAGGIO USATO.

$$\text{PRODUTTIVITÀ} = \text{LOC}/M \quad (M = \text{mesi uomo})$$

$$\text{QUALITÀ} = E/\text{LOC} \quad (E = \text{errori})$$

$$\text{COSTO UNITARIO} = \$/\text{LOC}$$

$$\text{LIVELLO DI DOCUMENTAZIONE} = PD/\text{LOC} \quad (PD = \text{quanto è documentato per linee di codice})$$



Complessità Ciclomatica

Misura la complessità del GRAFO del FLUSSO di controllo, in termini di numero di cammini lineariamente indipendenti che dal nodo PRINCIPALE raggiungono quello FINALE.

$$v(G) = e - n + 2p$$

Un valore elevato indica una struttura complessa

e = numero di archi del GRAFO
 n = numero di nodi
 p = numero di COMPONENTI CONNESSE

Function Points

Per fare ciò misure le funzionalità offerte del SW, senza entrare troppo nel dettaglio. Divide il SW in componenti e li misura.

Fornisce quindi un PARAMETRO ADIMENSIONALE dato che non dipende da un' unità di misura.

Si basa su un disegno logico del SW espresso in una forma qualsiasi, cosa che permette l'indipendenza dall'AMBIENTE TECNOLOGICO, e le possibilità di confrontare progetti diversi.

METRICHE ARCHITETTURALI

FAN-IN

Misura il numero di moduli che chiamano su un certo modulo.

FAN-OUT

Misura il numero di moduli che un certo modulo chiama.

Collaudo

Può misurare:

- La percentuale di copertura dei TEST-CASE
- $\frac{\text{n° FAILURES INDIVIDUATI}}{\text{LOC}}$
- $\frac{\text{n° FAULTS INDIVIDUATI}}{\text{LOC}}$

FAULT = DIFETTO

FAILURE = STATO o CONDIZIONE di non raggiungimento di un obiettivo.

$$AFP = COSTO - FP$$

↑
Adjusted
FP

VAF

I VAF si calcolano rispondendo ad un questionario dove ogni risposta ha un valore da 0 a 5.

La somma di questi valori, moltiplicata per 0,01, rappresenta i VAF.

Il VAF può andare da 0,65 a 1,35.

$$AFP = UFP * VAF$$

unadjusted
FP

tiene conto dei
REQ. FUNZIONALI
(quante cose
dobbiamo fare)

value adjustment
factor

tiene conto dei REQ.
NON FUNZIONALI e di
caratteristiche generali
del SISTEMA

UFP

Gli UFP si calcolano facendo le somme degli elementi di 5 indici:

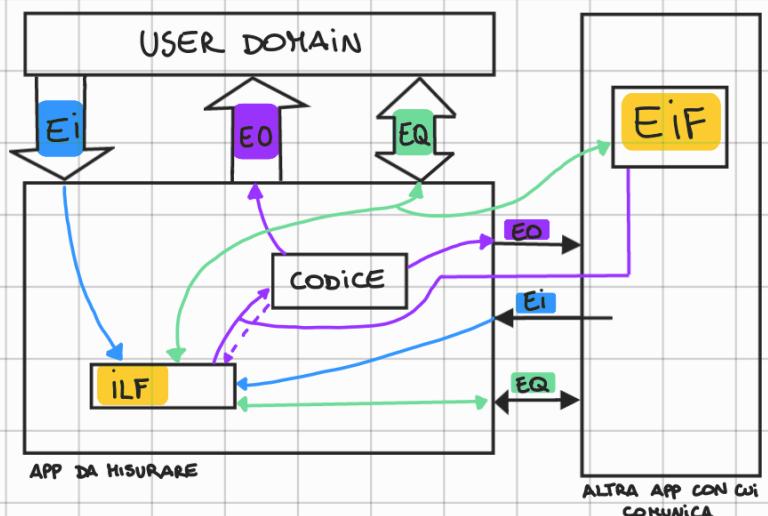
- 1. ILF (Internal Logical Files)
 - 2. EIF (External Interface Files)
 - 3. EI (External Input)
 - 4. EO (External Output)
 - 5. EQ (External Query)
- } DATI
- } TRANSAZIONI
SUI DATI

di ogni elemento si valuterà
poi la complessità.

La somma dei valori della
complessità, rappresenta gli UFP.

ILF: gruppo di dati o info di controllo
logicamente collegati e riconoscibili dell'utente, che sono
mantenuti all'interno dei confini
dell'APPPLICAZIONE.

EIF: stessa cosa dell'ILF ma è
mantenuto all'interno dei confini
di un'altra app, e viene REFERENZIATO dall'app che si sta misurando.



CONTEGGIO ILF E EIF

Ad ogni ILF e EIF viene assegnata una COMPLESSITÀ FUNZIONALE basata sul numero di elementi di tipo:

- DATI (DET) → campo unico e non ripetuto
- RECORD (RET) → sottogruppo di elementi di tipo DATI riconoscibili dell'utente in un ILF o EIF.
Se non ci sono sottogruppi, si conta il ILF o EIF come un RET.

Per calcolare la complessità si usa la seguente metrice:

| | 1-19 DET | 20-50 DET | 51+ DET |
|---------|----------|-----------|---------|
| 1 RET | BASSA | BASSA | MEDIA |
| 2-5 RET | BASSA | MEDIA | ALTA |
| 6+ RET | MEDIA | ALTA | ALTA |

E1: processo elementare dell'APP che elabora DATI o INFO di controllo provenienti da un'altra APP.

COMPITO PRINCIPALE → mantenere 1+ iFL e/o modificare il comportamento del sistema.

E0: processo elementare dell'APP che manda dati o info di controllo all'esterno del confine dell'APP.

COMPITO PRINCIPALE → presentare info all'utente attraverso una logica di processo diversa / in aggiunta al recupero di dati e info di controllo.

La logica di processo contiene almeno 1 FORMULA MATEMATICA, crea dati derivati e mantiene 1+ iFL o modifica il comportamento del sistema.

La logica di processo NON contiene FORMULE MATEMATICHE e NON crea DATI DERIVATI.

Nessun iLF è mantenuto durante l'elaborazione e il comportamento del sistema non è alterato.

FP NON PESATI

Il numero di FP non pesati per ciascuna funzione si ottiene usando le seguenti tabelle di conversione.

| VALORE COMPLESSITÀ FUNZIONALE | TIPI DI FUNZIONE | |
|-------------------------------|------------------|-----|
| | iLF | EiF |
| BASSA | 7 | 5 |
| MEDIA | 10 | 7 |
| ALTA | 15 | 10 |

FP
LOC

FP

- indipendente dal linguaggio

- basato sulle specifiche

- user oriented

- FP → LOC ✓

LOC

- dipendente dal linguaggio

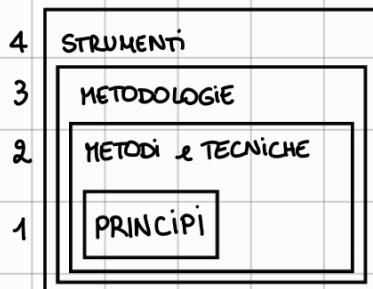
- basato sulla analogia (contatto a posteriori)

- design oriented

- LOC → FP ✓

PRINCIPI

1



- supportano l'applicazione di una METODOLOGIA.
- coordinano un insieme di METODI e TECNICHE consistenti tra loro, secondo un approccio comune.
- usati per applicare i PRINCIPI
- descrivono proprietà desiderabili del PROCESSO e dai PRODOTTI per lo SVILUPPO DEL SW.

PRINCIPI

2

- RIGORE E FORMALISMO: durante la codifica e la documentazione.

- SEPARAZIONE DEGLI INTERESSI: dividere i compiti e le responsabilità.



- MODULARITÀ: prima vengono trattati i dettagli dei singoli moduli e poi le relazioni tra di loro. Per costruire SW modulare ci sono 2 approcci: - BOTTOM UP → si trattano prima i singoli moduli e poi si compongono insieme.

ALTA COESIONE
BASSO ACCOPPIAMENTO

- TOP DOWN → prima si studia la scomposizione del problema in moduli, e poi i singoli moduli.

- ASTRAZIONE: consente di identificare gli aspetti fondamentali di un fenomeno, ignorandone i dettagli.

- ANTICIPAZIONE DEL CAMBIAMENTO

- GENERALITÀ: quando si cerca di risolvere un problema bisognerebbe cercare di capire quale è il problema più GENERALE che si nasconde dietro. Spesso il problema più generale è più semplice, e le sue soluzioni più RIUSABILE. Potrebbe tuttavia essere più COSTOSA.

- INCREMENTALITÀ: prevede che il PROCESSO DI PRODUZIONE del SW procede attraverso una serie di approssimazioni successive

modello CASCATA (INCREMENTALE)

Le ATTIVITÀ sono strutturate come una CASCATA LINEARE DI FASI

L'OUTPUT di una fase, diventa l'INPUT delle successive.

Ogni fase è divisa in SOTTO-ATTIVITÀ che possono essere eseguite contemporaneamente.

È tuttavia un MODELLO MOLTO DISPENDIOSO dove l'intero processo va pianificato in ANTICIPO.

È DOCUMENT-DRIVEN e una fase inizia solo dopo che la precedente è terminata.

Le varie fasi possono anche mandare FEEDBACK alle fasi strettamente precedenti o ad una fase antecedente qualsiasi.

VANTAGGI

- FORTE DISCIPLINA durante tutto il ciclo di vita del SW.
- Permette di stabilire subito gli obiettivi di consegna associati alle relative DATE.
- Ottimo per quando si devono conoscere a fondo le specifiche prima di iniziare la progettazione.

Svantaggi

- FASI RIGIDE, non sono quindi previste interazioni intermedie tra CLIENTE e SVILUPPATORI.
- I COSTI DI MANUTENZIONE aumentano in caso di INCOMPRESIONI o CAMBIAMENTI.
- EVENTUALI ERRORI potrebbero non emergere fino al rilascio del prodotto.
- Alcuni REQUISITI potrebbero non essere più attuali se passa molto tempo dall'inizio del progetto.

PROCESSI PER LA PRODUZIONE DEL SW

Come si organizza un lavoro di produzione?

Oggi lo sviluppo del SW è un'attività professionale che richiede una PIANIFICAZIONE.

MODELLO DEL PROCESSO SW

Rappresentazione semplificata del processo SW, che espone le STRUTTURE PRINCIPALI del processo ma non i dettagli.

MODELLO CODE AND FIX

Il progettato re alle prime svolte, prima scrive il codice e poi corregge gli errori. Tutto rende un piano.

MODELLO PLAN DRIVEN

Tutte le attività sono pianificate in anticipo. La produzione del SW può quindi essere scomposta in 4 macro ATTIVITÀ SPECIFICHE:

0. STUDIO DI FATTIBILITÀ: attività preliminare al processo di produzione che serve a trovare possibili soluzioni alternative, determinare i costi per lo sviluppo, ecc... Tipicamente si decide se:

Sviluppare un SW ex novo

Comprarlo da terzi

Abbandonare il progetto perché poco realistico

1. SPECIFICA:

è l'ANALISI DEI REQUISITI, vengono quindi definite le funzionalità del SW e fissati i vincoli operativi.

2. SVILUPPO:

generalmente suddiviso in ARCHITETTURA e IMPLEMENTAZIONE. Viene realizzato il SW con le specifiche del punto precedente.

3. CONVALIDA:

il SW viene TESTATO per accertarsi che corrisponde alle SPECIFICHE.

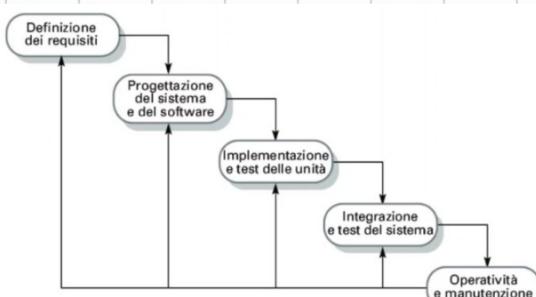
4. EVOLUZIONE:

il SW si evolve per soddisfare le necessità e i cambiamenti dei requisiti del committente.

(guarda il quaderno per approfondire)

MODelli

EsISTONO diversi MODELLI per organizzare queste fasi:



modello EVOLUTIVO (o INCREMENTALE)

Modello meno PLAN-DRIVEN e più AGILE.

Si produce subito una versione iniziale del SW che viene subito esposta agli utenti per ottenere FEEDBACK IMMEDIATI. Quindi avremo:

1. RILASCIO di FUNZIONALITÀ all' UTENTE
2. MISURA del VALORE AGGIUNTO per il CLIENTE
3. AGGIORNAMENTO sia del progetto che degli OBIETTIVI in base a quanto osservato.

} I CAMBIAMENTI FAONO PARTE dello SVILUPPO e si ricorre ad una CONSEGNA INCREMENTALE

- | VANTAGGI | SVANTAGGI |
|--|--|
| <ul style="list-style-type: none"> ● COSTO DEI REQUISITI e delle MODIFICHE RIDOTTO ● Più semplice ottenere FEEDBACK del cliente ● CONSEGNA INCREMENTALE: il cliente può usare il SW prima del rilascio finale | <ul style="list-style-type: none"> ● Non è economico produrre molte DOCUMENTAZIONE per ogni incremento del prototipo. È quindi più complesso da gestire. ● È necessaria una COSTANTE riORGANIZZAZIONE del CODICE perché la STRUTTURA DEL PROGRAMMA tende a degradarsi ad ogni incremento ed il codice si complica. |

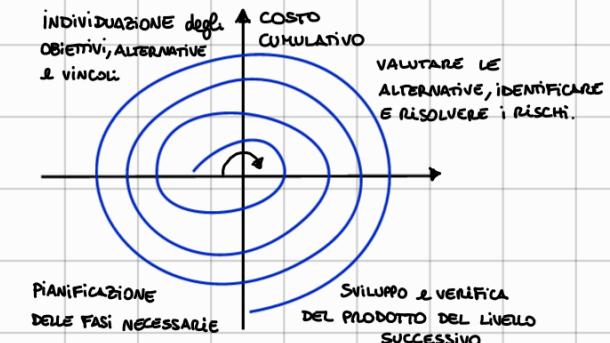
modello A SPIRALE

È un METAMODELLO, cioè è in grado di DESCRIVERE qualsiasi modello di processo di sviluppo (CASCATA, EVOLUTIVO,...) L'OBBIETTIVO è quello di fornire un QUADRO DI RIFERIMENTO per la progettazione dei processi al fine di minimizzare i RISCHI. La GESTIONE DEI RISCHI consiste nell' IDENTIFICARE i rischi in cui il SW può incorrere prima che diventino minacce serie.

Il modello è CIClico ed ogni ciclo della SPIRALE consiste di 4 FASI, ognuna delle quali rappresenta un quadrante del piano cartesiano.

IL RAGGIO della SPIRALE rappresenta il COSTO ACCUMULATO del processo fino a quel momento.

La DIMENSIONE ANGOLARE della SPIRALE rappresenta il PROGRESO DEL PROCESSO.



modello SVILUPPO AGILE

MANIFESTO

- INDIVIDI e INTERAZIONI sono più importanti dei PROCESSI e altri strumenti.
- SW FUNZIONANTE più importante della DOCUMENTAZIONE ESAUSTIVA.
- COLLABORAZIONE col CLIENTE più importante della NEGOZIAZIONE dei CONTRATTI.
- RISPONDERE al CAMBIAMENTO più che seguire un piano.

Il requisito più critico per le aziende che producono SW è quello di produrlo RAPIDAMENTE, soprattutto se è presente della CONCORRENZA.

I REQUISITI dei clienti, inoltre, diventano chiari solo dopo che il sistema è in UTILIZZO, quindi i processi di sviluppo PLAN-DRIVEN non sono adatti a questo scenario.

Questo metodo è l'ideale per lo sviluppo di prodotti PICCOLI o MEDI, e per uno SVILUPPO PERSONALIZZATO dove il CLIENTE viene coinvolto nel processo e dove ci sono pochi STAKEHOLDER che possono influire sul SW.



MINIMAL VIABLE PRODUCT → PRODOTTO MINIMO FUNZIONANTE, fornire subito all'utente un prodotto che ha un certo valore di utilizzo, così da evitare di costituire prodotti che i clienti non vogliono.
(è il PRODOTTO con il più ALTO RITORNO sugli INVESTIMENTI rispetto al RISCHIO)

EXTREME PROGRAMMING → SVILUPPO INCREMENTALE supportato attraverso piccole e frequenti RELEASE DI SISTEMA. I REQUISITI sono basati su semplici SCENARI per decidere le funzionalità da includere in un incremento. Prevede, inoltre, GRUPPI DI PROGRAMMAZIONE PICCOLI (coppie), un POSSESSO COLLETTIVO del CODICE SORGENTE e un ritmo di sviluppo COSTANTE.

STORIA UTENTE → scenari d'uso in cui potrebbe trovarsi l'utente. Viene redatta una STORY CARD che raccoglie le esigenze dell'utente. Ogni STORY CARD è divisa in TASK e vengono stimate le RISORSE e gli SFORZI richiesti per implementare ciascun TASK.
Le storie vengono ordinate per PRIORITÀ del cliente.
PROBLEMI: - le storie potrebbero essere incomplete/poco veritieri.
- se i req. variano, le STORIE non implementate possono cambiare.

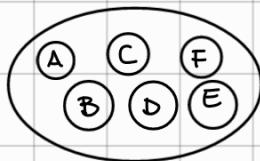
REFRACTING → RISTRUTTURAZIONE del codice che ne migliora la qualità, la struttura e la leggibilità, contrastando il DETERIORAMENTO.
l'extreme programming preferisce il REFRACTING alla progettazione che anticipa il cambiamento.

TEST → possono essere scritti prima del codice così da risolvere subito eventuali AMBIGUITÀ e omissioni nelle specifiche. Il PROGRAMMA è considerato "FINITO" quando supera questi test.
Ogni TASK genera 1+ test, che sono AUTOMATIZZATI, quindi lo sviluppo non può proseguire finché non sono stati superati tutti. IL CLIENTE aiuta a sviluppare i TEST di accettazione delle storie.

METODO SCRUM → si occupa dell'ORGANIZZAZIONE del TEAM e della PIANIFICAZIONE FORMALE di un PROGETTO AGILE, in modo che il tutto sia ben visibile al MANAGER per fargli capire cosa sta accadendo.

↑
RIUNIONE GIORNALIERA

PRODUCT BACKLOG



SPRINT BACKLOG

| TO DO | WIP | DONE |
|-------|-----|------|
| (E) | (B) | (A) |
| (C) | | |
| (D) | | |

SVANTAGGI SVILUPPO:
AGILE

Dei cicli di rilascio brevi consentono meglio:

TEMPESTIVITÀ **FLESSIBILITÀ** **FIX RAPIDI** **RELEASE STABILI**

Ecco perché nasce DEVOPS, un INSIEME DI PRATICHE e CAMBIAMENTI di PROCESSO AUTOMATICI, al fine di automatizzare il rilancio del SW rispetto alla CATENA di PRODUZIONE, ottenendo un SW di migliore QUALITÀ e più sicuro in TEMPI MINORI.

I TEAM di LAVORO sono ONE TEAM INTERFUNZIONALI piccoli, così che possono mantenere il FOCUS su un aspetto specifico del prodotto.

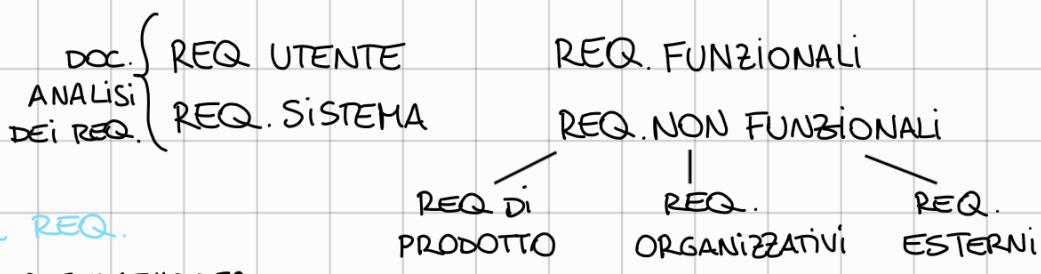
- L'INFORMALITÀ di questo metodo è spesso INCOMPATIBILE con l'approccio LEGALE nella definizione dei contratti.
- Non è infatti possibile definire i REQ. definitivi in anticipo e ci si deve basare su contratti in cui è il TEKNO di sviluppo ad essere pagato, piuttosto che i REQ.
- Poco adatto alla MANUTENZIONE
- Pensato per piccoli TEAM che lavorano fisicamente insieme.
- NON SCALABILE:
 - TEAM di solito sono DISTANTI GEOGRAFICAMENTE
 - REGOLE e NORME ESTERNE potrebbero chiedere DOCUMENTI e specifici REQ. di conformità.
 - TEMPI DI SVILUPPO LUNghi
 - MOLTI STAKEHOLDER con obiettivi diversi. Non è possibile coinvolgerli tutti.

Sviluppo e produzione sono fusi in un'unica unità in cui i tecnici sono attivi lungo tutto il ciclo di vita del SW. Prassi DevOps:



Ingegneria dei Requisiti

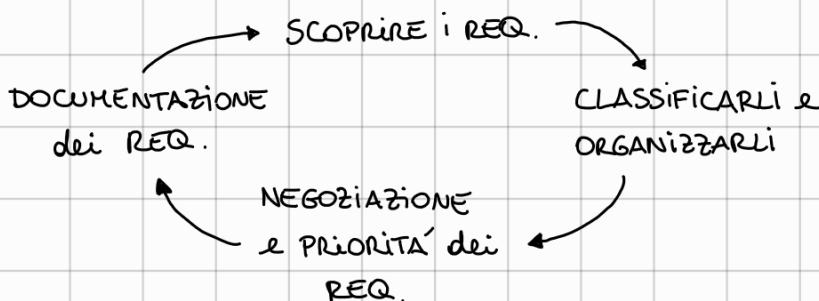
È il processo di RICERCA, ANALISI, DOCUMENTAZIONE e VERIFICA dei REQ. A seconda del loro livello di descrizione, possiamo avere:



DEDUZIONE e ANALISI dei REQ.

si cerca di capire intervistando gli STAKEHOLDER, i SERVIZI, PRESTAZIONI, VINCOLI HW, ecc... che un SISTEMA dovrebbe avere.

Bisogna distinguere tra DESIDERI e BISOGNI del committente.



Si può usare
L'ETNOGRAFIA.

SPECIFICA dei REQ.

Si scrivono i REQ. dell'UTENTE e del SISTEMA in un DOCUMENTO.

Vengono descritti solo i VINCOLI OPERATIVI del sistema e il suo comportamento esterno, senza toccare aspetti di progettazione o implementazione.

Tutto ciò in LINGUAGGIO NATURALE, introducendo un GLOSSARIO, oppure con LINGUAGGI STRUTTURATI per mitigare le fonti AMBIGUITÀ del linguaggio naturale.

I CASI D'USO utilizzano una NOTAZIONE GRAFICA e un TESTO STRUTTURATO per descrivere le interazioni tra UTENTE e SISTEMA.

GESTIONE DELLA CONFIGURAZIONE → gestire un sistema su im evoluzione (politiche, processi, e strumenti). Prevede 4 ATTIVITÀ:

1. CONTROLLO delle VERSIONI
2. COSTRUZIONE DEL SISTEMA
3. GESTIONE delle MODIFICHE
4. GESTIONE delle RELEASE

DESIGN PATTERN

Visto che la programmazione ad oggetti può risultare difficile, ci sono delle soluzioni (dei pattern) comuni a problemi pratici.

Un **design pattern** è quindi una struttura di soluzione che raccoglie l'esperienza dei progettisti.

Elementi essenziali del design pattern:

- nome.
- problema che risolve.
- soluzione, generalmente descritta con un diagramma UML.
- conseguenze.

MVC

Un esempio di design pattern è la soluzione al problema di sincronizzazione tra le viste di un modello (excel): vedere i dati sottoforma di grafico, di formula, di numero... se cambio i dati in una di queste viste, si aggiornano in tutte le altre viste.

Il design pattern utilizzato in questo caso è il **Model, View e Controller (MVC)**.

- **Model:** valore del dato vero e proprio, indipendente dalle rappresentazioni sullo schermo e dalla modalità di input dei dati da parte dell'utente.
- **View:** rappresentazione dell'oggetto sullo schermo dell'utente. Una view ottiene i dati dal Model. Possono esistere più views dello stesso modello.
- **Controller:** insieme di regole che stabiliscono le reazioni della presentazione sullo schermo in relazione all'input dei dati da parte dell'utente. Ogni view ha un controller che deve ricevere gli input dell'utente e tradurre gli eventi in richieste di servizio per il model o la vista a cui è associato. Intercetta quindi la volontà dell'utente di voler cambiare i dati.

Quindi, quando l'utente vuole cambiare un dato, il Controller manda la richiesta al Model che decide se poter effettuare l'operazione o meno. Se l'operazione viene effettuata, il Model comunica a tutte le viste la modifica, e le View richiedono al Model il nuovo dato, aggiornando la presentazione a schermo del tutto.

Esercizio Testing:

Si consideri il seguente codice e si forniscano i casi di test minimali strutturali che realizzano lo statement coverage, il condition coverage e il path coverage.

```
int otto (int x, int A[]){
```

```
    int z=0;           //c1
    if (x > 0 && A[0] > x) //c2
        z=A[0];         //c3
    if(A[0] < 0)       //c4
        z++;
    else z--;          //c6
    return z;          //c7
```

Dobbiamo quindi dare un input che ricopre tutti i casi.

STATEMENT COVERAGE:

| x | A[] | output | statement coperti |
|---|-------------|--------|-------------------|
| 1 | A[] = [10] | 9 | c1,c2,c3,c4,c6,c7 |
| 1 | A[] = [-10] | -1 | c1,c2,c3,c5,c7 |

}

CONDITION COVERAGE: deve considerare le condizioni composte ($x > 0 \ \&\& \ A[0] > x$)

| x | $A[]$ | Output | $x > 0$ | $A[0] > x$ | |
|-----|---------------|--------|---------|------------|--|
| 1 | $A[] = [10]$ | 3 | 1 | 1 | $\rightarrow 1 > 0 \ \& \ A[0] > 1$. |
| 1 | $A[] = [-10]$ | -1 | 1 | 0 | \rightarrow qui $A[0]$ non è $> x$. |
| -1 | $A[] = [-2]$ | -1 | 0 | 0 | |
| -1 | $A[] = [0]$ | -1 | 0 | 1 | |

PATH COVERAGE: abbiamo 4 path

| | | | |
|-----|----------------------|----------|---------------|
| P1: | if1 vero, if2 falso | $x = 1$ | $A[] = [10]$ |
| P2: | if1 vero, if2 vero | $x = 1$ | $A[] = [-10]$ |
| P3: | if1 falso, if2 vero | $x = -1$ | $A[] = [-2]$ |
| P4: | if1 falso, if2 falso | $x = -1$ | $A[] = [5]$ |

| | | Campo di applicazione | | |
|---------------|--|--|---|-----------------|
| | | Creational (5) | Structural (7) | Behavioral (11) |
| Class | Factory method | Adapter (Class) | Interpreter Template Method | |
| Relazioni tra | Object  Abstract Factory Builder Prototype  Singleton | Adapter(Object) Bridge  Composite Decorator Facade Flyweight  Proxy | Chain of Responsibility Command Iterator Mediator Memento  Observer State Strategy Visitor | MVC |

Design Pattern Creazionali (Singleton e Abstract Factory)

Singleton

Problema che risolve: assicurare l'esistenza di un'unica istanza di una classe.

L'istanza deve essere accessibile dai client (i programmi) in modo noto, e deve essere estendibile con ereditarietà, senza che i client modifichino il loro codice.

Conseguenze: controllo completo di come e quando i client accedono all'interfaccia.

○ Struttura

Definisce `getInstance` che permette l'accesso all'unica istanza. È responsabile della creazione dell'unica istanza



```

/*
 * Implementazione Java
 * "naive"
 */
public class Singleton {
    private static
        Singleton instance;
    private Singleton() {
        /* Corpo vuoto */
    }
    public static Singleton
        getInstance() {
        if (instance == null) {
            instance =
                new Singleton();
        }
        return instance;
    }
}
  
```

Abstract Factory

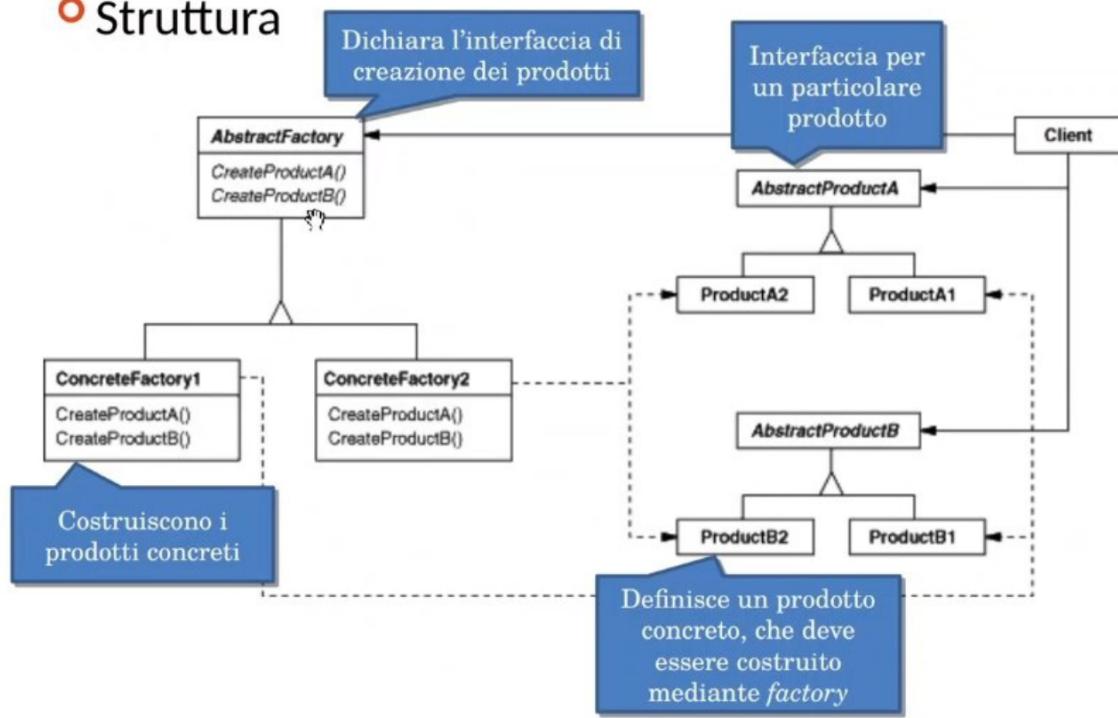
Fornisce un'interfaccia per creare famiglie di prodotti senza specificare classi concrete. Questo per avere un'applicazione configurabile con diverse famiglie di componenti.

Per esempio per l'implementazione di un'App che usa un toolkit grafico (bottoni, finestre, menù a tendina, ecc...), ci sarà bisogno della dichiarazione delle classi concrete degli oggetti Bottone, Finestra, ecc...

Il problema è che se voglio che la mia App sia portatile, e che quindi funzioni anche su Linux, ad esempio, dovrò fare la dichiarazione di altre classi Bottone, Finestra, ecc... specifiche per quel sistema operativo.

L'idea di questo pattern è quella di definire una classe astratta Factory che definisce le interfacce di creazione astratte. Sarà poi una classe Factory concreta (figlia di quella astratta) a creare gli oggetti specifici.

○ Struttura



Conseguenze:

i client manipolano
unicamente le
interfacce, i nomi
dei prodotti non li
conoscono.

C'è inoltre una
maggiore
semplicità
nell'utilizzo di una
famiglia di prodotti
perchè la Factory
concreta appare
solo una volta nel
programma.

Design Pattern Strutturali (Object Adapter, Proxy e Composite)

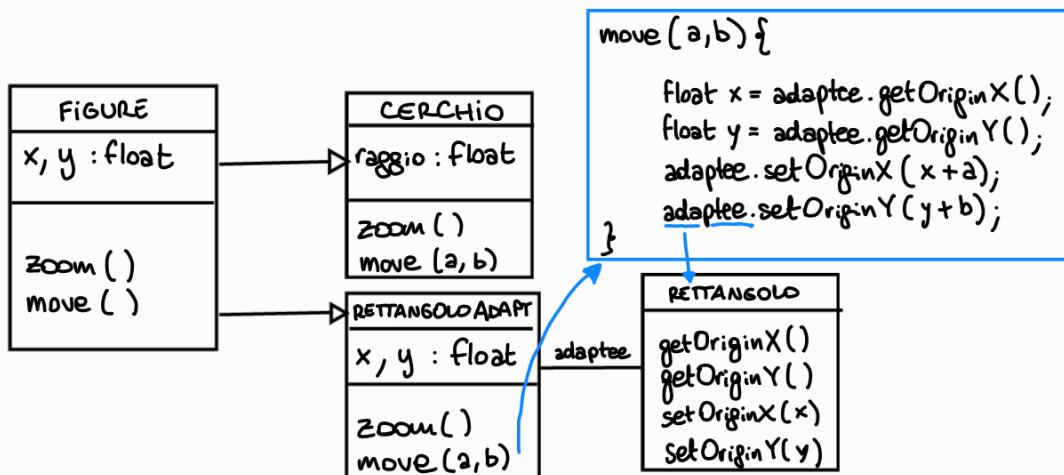
Hanno lo scopo di risolvere i problemi relativi alla strutturizzazione di classi e oggetti, consentendo il riutilizzo degli oggetti esistenti tramite un'interfaccia adatta per gli utilizzatori. (Integrazioni tra librerie e componenti diverse)
Sfruttano l'ereditarietà e l'aggregazione.

Adapter: converte l'interfaccia di una classe in un'altra. Viene quindi definita una classe adattatore che adatta le interfacce per l'ereditarietà o composizione, e che può anche fornire alla classe adattata funzionalità che questa non possiede.

Se non si può convertire l'interfaccia attraverso l'ereditarietà, si deve usare il pattern **Object Adapter**: il quale permette ad un Adapter di adattare più classi (Adaptee e le sue sottoclassi), e non permette di modificare le caratteristiche dell'Adaptee (un oggetto Adapter non è sottotipo dell'Adaptee).

Per esempio se ho una classe già fatta "Rettangolo" (Adaptee), che voglio utilizzare ma che però non si adatta alla mia classe più generale "Forme", posso utilizzare una classe Adattatore "RettangoloAdapter" che utilizza un oggetto Rettangolo (wrapping, perché la classe incapsula l'oggetto Rettangolo).

L'Adapter è il figlio della classe che voglio adattare, quindi RettangoloAdapter è figlio di Forme.



Nel caso del **Proxy**, invece, lo scopo è quello di fornire un surrogato di un altro oggetto di cui si vuole controllare l'accesso, per rinviare il costo di creazione di un oggetto all'effettivo utilizzo. Anche in questo caso il Proxy ingloba l'oggetto.

Possiamo avere diversi tipi di Proxy:

- **Remote Proxy:** rappresentazione locale di un oggetto che si trova in uno spazio di indirizzi differente. Questo permette di nascondere dove un oggetto risiede.
- **Virtual Proxy:** creazione di oggetti complessi on-demand, ma il sistema può funzionare come se l'oggetto fosse effettivamente presente (ottimizzazioni).
- **Protection Proxy:** controllo degli accessi all'oggetto originale.
- **Puntatore "intelligente":** gestione della memoria.

Per esempio se voglio caricare una foto in Word, Word mi farà vedere la pagina che sto editando costituita da oggetti di tipo testo, immagine, ecc...

L'immagine che voglio vedere è di tipo immagine, e la chiameremo Subject.

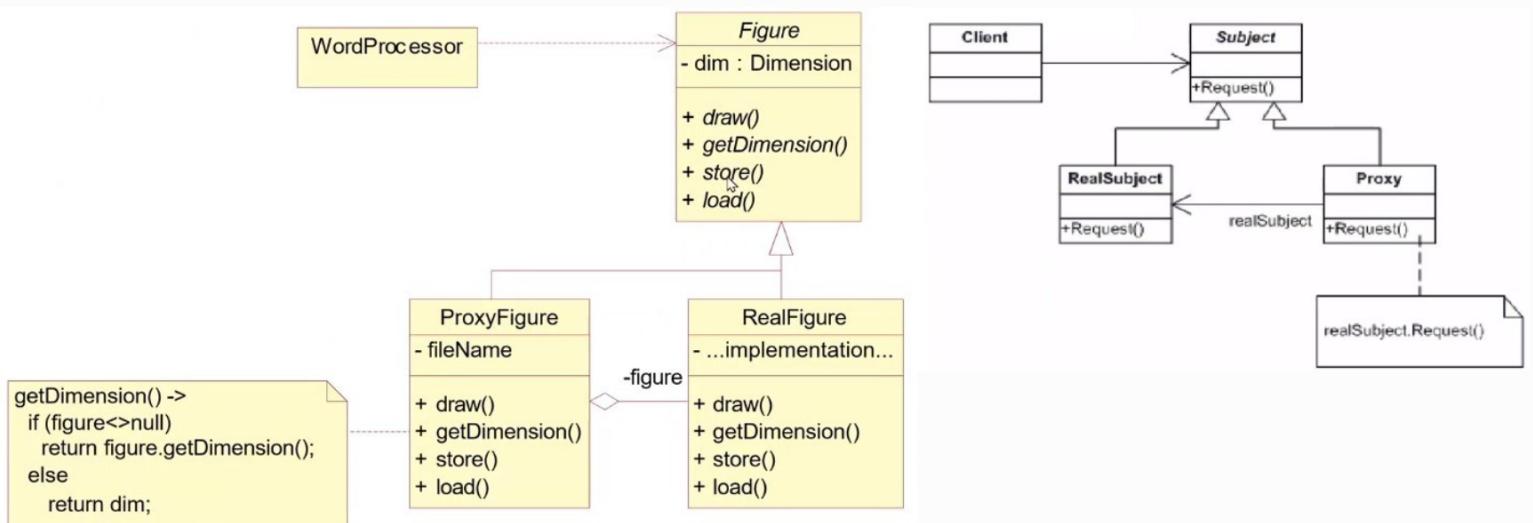
Questo Subject ha due sottoclassi: RealSubject, cioè l'immagine vera e propria, e Proxy.

La classe Proxy utilizzerà al suo interno un oggetto di tipo RealSubject.

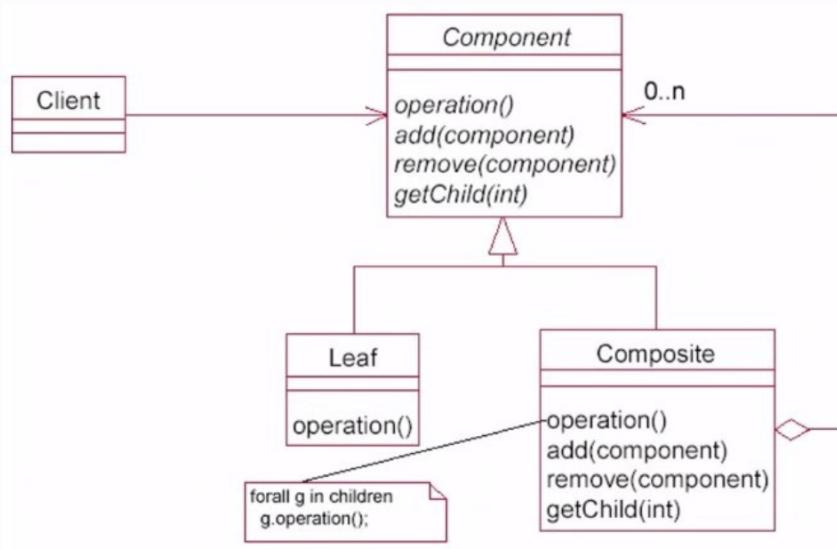
Quando il Subject generico viene chiamato, il metodo non viene eseguito subito in RealSubject, ma nel Proxy che implementa tutti i suoi metodi.

Questo perchè il caricamento effettivo dell'immagine richiede più tempo, e viene fatto solo se l'immagine deve essere effettivamente vista dall'utente.

La classe Proxy, quindi, filtra tutte le chiamate ai metodi di RealSubject, e decide quando far passare tali chiamate.



Composite: fornisce una possibilità di comporre oggetti in strutture ad albero che rappresentano gerarchie intero-parte. Consente ai client di trattare oggetti singoli e composti in modo uniforme, e di rendere la gerarchia intero-parte meno complessa.



Design Pattern Comportamentali (Observer)

L'Observer è la forma più generica del MVC.

Definisce una dipendenza 1..n (uno a molti) tra oggetti, riflettendo la modifica di un oggetto su i suoi oggetti dipendenti. Questo per mantenere la consistenza fra oggetti (modello e viste ad esso collegate). E' praticamente il meccanismo chiamato Publish-Subscribe, ovvero un oggetto si iscrive a ricevere notifiche sul cambiamento dello stato di un altro oggetto.

Nel MVC c'erano i Model, le View e i Controller, mentre nell'Observer ci sono solo due categorie di oggetti: i **Subject** (Model) e gli **Observer** (View e Controller).

E' il Subject che attiva l'aggiornamento delle viste.

Gli Observer vengono notificati solo se è avvenuto un cambiamento sul dato specifico per cui si sono iscritti, non per altri cambiamenti.

Conseguenze: accoppiamento astratto tra Subject e Observers (cioè i Subjects non conoscono il tipo concreto degli Observers), comunicazione Broadcast con libertà di aggiungere Observers dinamicamente, e aggiornamenti non voluti si ripercuotono a cascata su tutti gli Observers. Gli Observers, inoltre, non sanno cosa è cambiato nel Subject.

Push Model: il Subject conosce i suoi Observers.

Pull Model: il Subject invia solo la notifica

○ Struttura

