Appunti Ingegneria del software

Dario Di Meo

Gennaio 2021

Indice

1 Introduzione

l'ingegneria del software è la materia che introduce i *processi di sviluppo software* che risolvano i problemi della produzione del software.

Quindi avere un approccio ingegneristico al problema, applicando metodi quantificabili che ci permettono di risolvere i problemi classici che attanagliano la produzione.
l'**IEEE** dice che:

> l'Ingegneria del software è l'applicazione di un approccio discipinato, qauntificabile e sistematico allo sviluppo, alla progettazione e alla manutenzione di un Software.

1.1 Cos'è un software

Secondo l'**IEEE** un software è:

> *Un Software è un insieme di **Programmi**, **procedure**, **regole** e ogni tipo di **documentazione** realtiva al suo funzionamento, di un sistema di eleaborazione dati*

Quindi un software non e' solo il codice, ma anche tutte le regole e soprattutto la *documentazione*.
Di particolare importanza è la differenza tra software amatoriale e professionale.

1. software amatoriale:
   - Non sarà usato da altre persone
   - Non dobbiamo soddisfare dei requisiti
   - Ci sara un solo sviluppatore
2. software professionale:
   - Sarà usato da altre persone
   - Deve soddisfare dei requisiti di affidabilità e di qualità
   - Documentazione necessaria

Quindi è evidente che il processo di sviluppo di un software professionale è molto più complesso. Quindi necessitiamo di metodi rigorosi per regolare questo processo complesso.

Un ulteriore differenza si può fare sul *Tipo di software*:

1. **software commerciali:** ad esempio il pacchetto Office, quindi applicazioni così dette "on the shelf"

2. **software shareware:** a prova temporale, ad esempio le cosiddette evaluation copy

3. **software freeware:** gratuito, con o senza codice sorgente. Rimangono i diritti dell'autore

4. **software public domain:** pubblico senza diritti

Quali sono allora le fasi fondamentali per lo sviluppo di un software?

1. **Specifica:** Definizione dei requisiti e dei vincoli di progettazione (analisi e specifica).

2. **Sviluppo:** Progettazione della soluzione e poi l'implementazione in codice.

3. **Testing:** Provare il codice in cerca di errori, oppure verificare che i requisti siano soddisfatti(verifica e convalida).

4. **Evoluzione:** modifica del software per addattarlo a nuovi requisti.

1.2 Il manifesto della Crisi: la nascita del Ing. del Software

Nel 1968, la produzione software era completamente allo sbando. La scrittura del codice non era regolamentate, era quasi un processo "artistico".

- Quasi sempre c'erano incomprensioni tra sviluppatori e cliente.
- Si finiva spesso Out Of Budget
- Non si consegnava il prodotto, oppure si superavano le scadenze fissate
- Software non di qualità

Quindi da queste problematiche, si sente la necessità di un approccio ingegneristico al problema. Nacque allora l'***Ing. del Software***

1.3 Il problema dei costi

Con il passare degli anni, abbiamo notato un incremento enorme del prezzo dei Software.
In passato il costo maggiore, era dato dai componenti **Hardware**, adesso i componenti costano molto poco a differenza dei Software.
Da Dove vengono questi costi?
Questi costi vengono principalmente dalle fasi di **Sviluppo** e **Manutenzione**

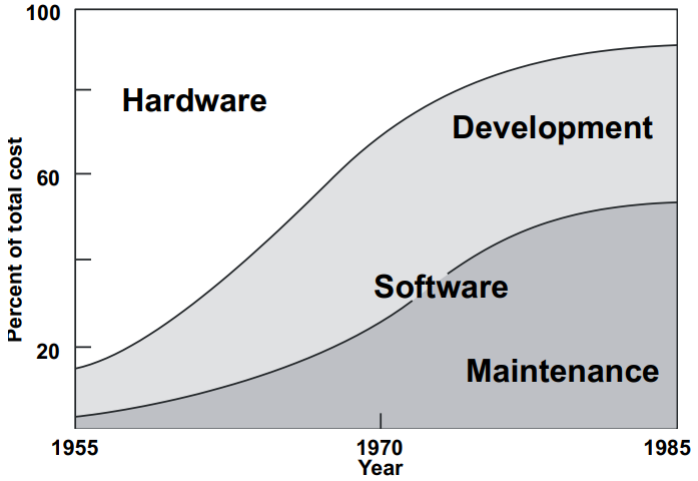

Figura 1: Costi negli anni

1.4 Problemi di Qualità

Vediamo adesso quali sono le caratteristiche di un software di **qualità**. Un software per essere di qualità deve essere:

- **Manutenibile:** Deve essere facile evolverlo, quindi supportato da una documentazione e codice scritto in maniera più astratta.
- **Affidabile:** Sebbene sia praticamente impossibile che un software si privo di errori, la probabilità che avvengano deve rimanere bassa. Caratteristica fondamentale per Software critici
- **Efficiente:** Deve soddisfare delle richieste in tempi utili(ad esempio i sistemi real-time necessitano tempi precisi e brevi)
- **Accetabile:** Deve essere accettato dagli utenti. Spesso troviamo software criptici difficilmente comprensibili dagli utenti. Per accetabili inoltre si intendono software anche che soddisfano le richieste fatte.

Spesso queste qualità sono difficili da verificare, in quanto si palesano solo quando il software viene usato, quindi nella fase operativa.

2 Ciclo di vita di un Software

Abbiamo detto nell'introduzione che un processo di sviluppo di un software deve contenere le fasi fondamentali di: **specifica, sviluppo, convalida ed evoluzione.** Compito del Ing del Software di regolare queste fasi del processo software quindi introdurre un **Modello di processo**.

> Un **Modello di Processo** non è altro che la descrizione dell' ordine di esecuzione di queste attività, e stabilire quali sono i criteri da verificare per poi passare alla fase successiva.

Quindi vogliamo utilizzare dei Modelli, che semplificano e soprattutto regolano il processo di sviluppo software[1].

2.1 Attività fondamentali nel ciclo di vita

Introduciamo adesso le **Attività fondamentali** per lo sviluppo software.

- **Studio di fattibilità:** Fase che produce un documento con la definizione dei costi/benefici, i tempi di consegna e le modalità di sviluppo
- **Specifica dei requisiti:** Fase che produce un documento che esplicita **COSA** il software dovrà fare e **NON COME**. quindi in questa fase non ci intressa la soluzione ma solo i requisiti.[2] L'oggetto prodotto e' il **DSR** (Documento di specifica dei requisiti) ed eventualmente il **PTS** (Piano di test del sistema).
- **Progettazione:** Progettare una soluzione al problema. La progettazione avviene a vari livelli di dettaglio
  - architettura generale: (HW e SW)Quindi i vari componenti (moduli) utilizzati e la funzionalità di ognuno
  - architettura dettagliata: descrizione dettagliata di ogni modulo. [3]
  - DSP (documento di Progetto): Documento che descrive i componenti (hw e sw) descrivendo il loro comportamento, le motivazioni delle scelte prese ed eventualemente future richieste.

[1] Il processo ovviamente dipende dal tipo di software, data la grande eterogeneità dei software

[2] Fase fondamanetale, errori in questa fase, corrompono tutte le successive. per torvarci infine un software che non soddisfa i requisiti del cliente

[3] Ad esempio in un approccio Client Server prima definiamo i moduli principali (client e server) poi andiamo nel dettaglio su ognuno.

- **Produzione:** Implementazione effettiva della soluzione progettata, quindi comprende:
  - codifica
  - documentazione del codice
  - testing del software
- **Manutenzione:** Dopo il rilascio del software, lo sviluppo non è finito. Esistono i seguenti tipi di manutenzione del software:
  - Correttiva: correggere errori presenti nel codice, sfuggiti dalla fase di testing.
  - Adattiva: addattare il nostro software ad un altra situazione (ad esempio nuovi S.O. , o cambio lingua).
  - Perfettiva: Perfezionare il software, per migliorarne la qualità (muntenibilità, affidabilità, efficienza e accetabilità).

2.2 Modelli di Processo

Abbiamo descritto nella pag. 6, la definizione di **Modello di Processo**. Ricapitolando, un modello di processo descrive l'ordine di esecuzione delle fasi fondamentali e i criteri da verificare alla fine di ogni fase.

Adesso introduciamo i principali modelli di processo più comuni. In passato prima della creazione dell'ing del software si usava il metodo **Code And Fix**.

Consisiteva nel produrre codice e in seguito correggere gli errori presenti. È una soluzione primitiva, non adatta a progetti di media/grande dimensione.

2.2.1 Modello a cascata: Waterfall

Il modello **Waterfall** è il modello più semplice. Il concetto principale di questo modello è che ogni fase può essere eseguita solo alla fine della precedente. Ogni fase alla sua fine, detta **Milestone**, produce un documento fondamentale per le fasi successive detto **deliverables** , per questo motivo è detto **guidato dalla documentazione** (Approccio altamente burocratico).

Può essere descritto da tre aggettivi:

- **Monoliticità**: Il progetto è completato solo alla fine di tutte le fasi
- **Rigidità**: Si assume che i requisiti siano **Congelati** nella fase di specifica dei requisiti
- **Linerità**: È lineare segue un modello di processo semplice, si parte da uno step se ne va in un altro. Non c'è parallelizzazione

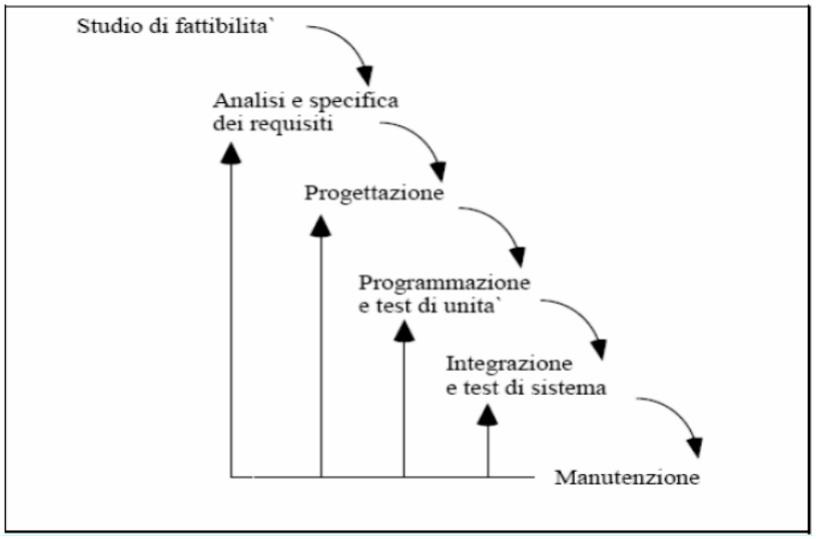

Figura 2: Modello a Cascata

Questo modello soffre di vari problemi tra cui:

- **Suddivisione dei compiti:** Si necessitano di figure che partecipano alle singole fasi, con il conseguente congelamento delle attività nei livelli sottostanti
- **Non modificabile:** Le modifiche nelle fasi successive, possono essere risolte solo alla terminazione, nella fase di manutenzione
- **Burocraticità:** Il modello dipende completamente dai documenti prodotti (i deliverables). Quindi rallentando di molto il processo
- **Stima dei costi:** È difficile effettuare una stima dei costi
- **Manutenzione:** Per la rigidità, siamo obbligati ad una manutenzione frequente e sostanziosa, aumentando anche di molto il costo in termini economici e temporali.

2.2.2 Modello a Retroazione a Cascata

È una variante del Modello Waterfall che introduce il concetto di **retroazione**.

Per retroazione indichiamo la possibilità di poter *tornare indietro*. Quindi ad ogni fase introduciamo due ulteriori **sotto-fasi** quella di **convalida** e di **verifica**.

- **Convalida:** Stabilire se il prodotto precedente è conforme alle sue specifiche
- **Verifica:** Verificare se il prodotto precedente è appropriato (è un parere soggettivo, non dipende dalle specifiche)

Per questo motivo è detto anche modello V & V a feedback.

Sebbene risolve il problema della monoliticità, del modello Waterfall, non può anticipare i cambiamenti. Quindi rimane sempre un modello lento, dovuto a questo overhead extra delle sotto-fasi di V & V.

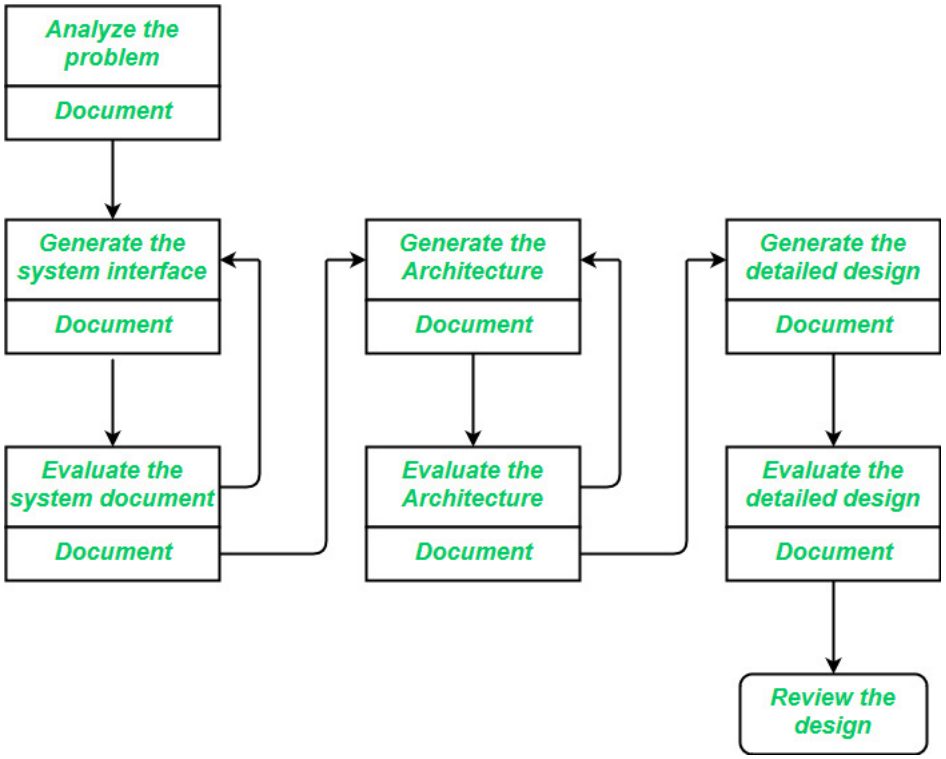

Figura 3: modello V&V a feedback

2.2.3 Modello a V

Il modello a V, è ampiamente utilizzato nei Software dei sistemi **Embedded** e nei sistemi di **controllo**. È ampiamente basato sul **Testing**, in quanto è spesso richiesto un alto livello di affidabilità. Essendo un modello **testing driven**, è necessario un PTS (piano di testing).

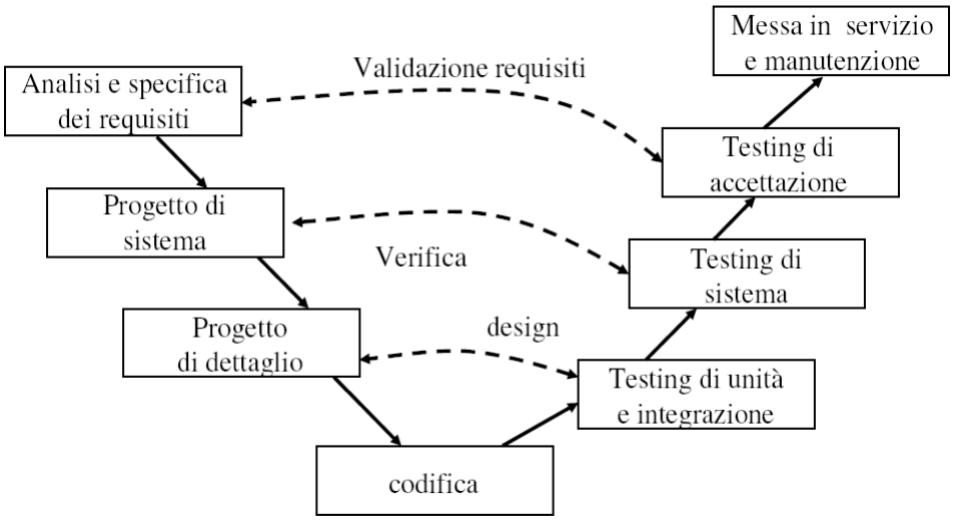

Figura 4: modello a V

Come vediamo dalla figura è un modello a doppia **retroazione**. E notiamo che la validazione è l'errore più costoso, in quanto ci obbliga a tornare alla fase di analisi e specifica dei requisiti.

Particolare enfasi, in questo modello, viene data all'**Hardware**.

Come per il modello a cascata e la sua variazione (V & V), viene usato per software in ambienti poco soggetti a cambiamenti.

2.3 Modelli Evolutivi

i modelli evolutivi si basano sulla produzione di una versione iniziale (Prototipo), questa versione verrà poi perfezionata attraverso varie versioni.

Prototipo

> Un Prototipo è un modello dell' applicazione, il cui obbiettivo è quello di ricevere feedback per affinare i requisiti.

Quindi utilizziamo i prototipi per raccogliere feedback dagli stakeholders. Questo permette un approccio più diretto col cliente, che diventa parte più integrante del progetto. Così facendo miglioriamo i requisiti ed eventualmente li **completiamo**.

I modelli Evolutivi che si basano sui prototipi, si dividono in base al utilizzo di esso:

- Prototipo **usa e getta**
- Prototipo **evolutivo**

2.3.1 Sviluppo con prototipo usa e getta

Con il prototipo usa e getta, si realizza una prima implementazione imperfetta e incompleta del software, con l'unico scopo di accertare la fattibilità e validare i requisiti. Poi il modello viene **gettato**.

Quindi si basa sul approccio "**Do It Twice**"

Questo approccio è ampiamente utilizzato in **software house con una certa maturità**. Questo perchè il prototipo è spesso creato con pezzi di codice già scritti, quindi è facile e veloce realizzarne uno.

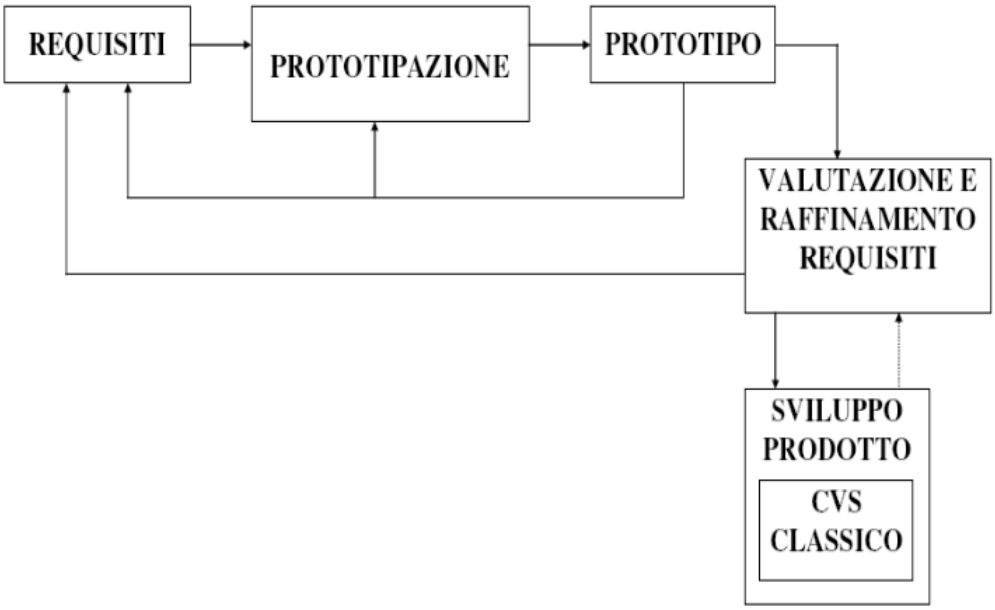

Figura 5: modello prototipale usa e getta

La creazione del prototipo può avvenire in varie fasi, sia in quella di **specifica dei requisiti** sia nella fase di **design**.

2.3.2 modello con prototipo evolutivo

I modelli con prototipo evolutivo, a differenza dei modelli con prototipi "usa e getta", non gettano il prototipo ma lo raffinano anche iterativamente, fino al prodotto finale.

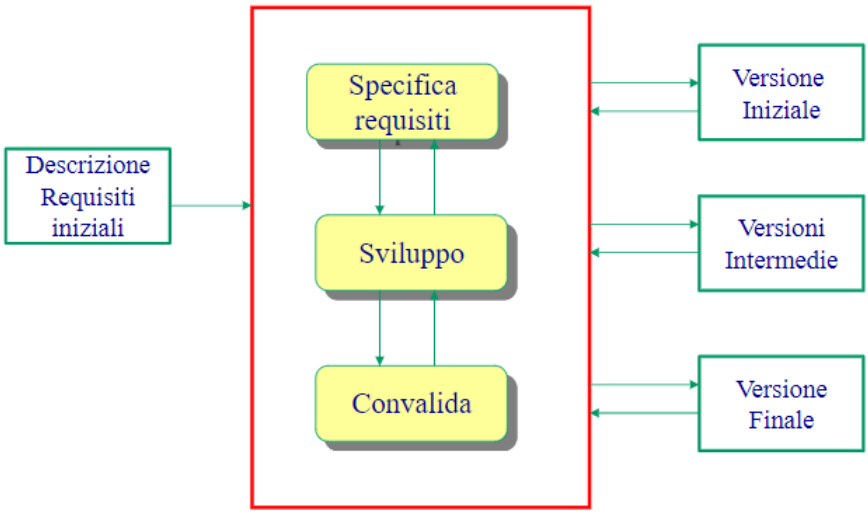

Figura 6: modello evolutivo

- vantaggi
  - sviluppo veloce
  - molti feedback dal cliente
  - cambio dei requisiti più veloce
- svantaggi
  - minor qualità dovuta al rattoppare il prototipo
  - pessima documentazione o assente
  - si necessità di programmatori con esperienza in linguaggi di prototipazione

2.4 Modello incrementale

Il modello incrementale, si basa sul rilascio ad **incrementi**.

Quindi il software non è rilasciato al completamento del software con tutti i requisiti soddisfatti , ma si rilascia ad incrementi, cioè a **step**. Quindi si associano dei livelli di **priorità** ai vari requisiti , e si sviluppano prima gli incrementi relativi a quelli con priorità maggiore.

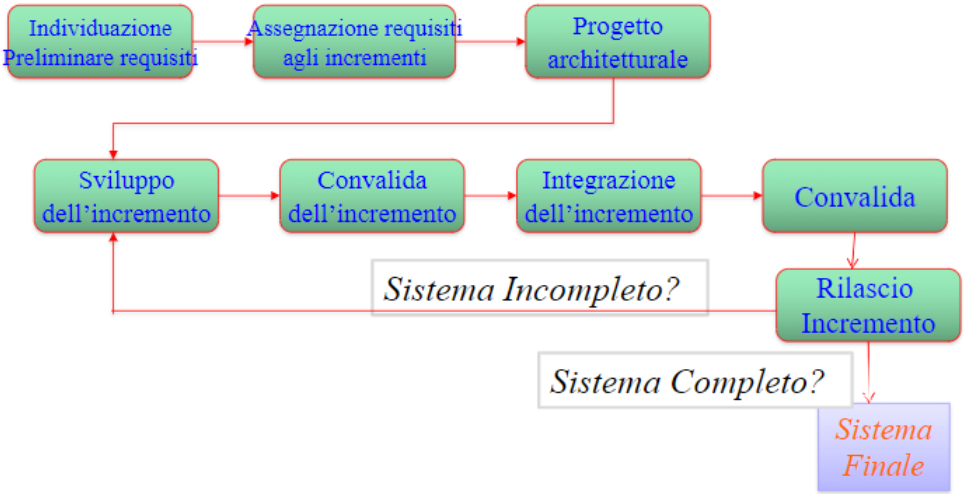

Figura 7: modello incrementale

I primi incrementi, fungono da veri e propri prototipi, che possono essere col tempo evoluti,

Tra i vantaggi abbiamo:

- Il cliente vede prima i risultati, risultando più soddisfatto
- Si riduce il rischio di fallimento del intero progetto
- I servizi con priorità maggiore vengono testati in maniera più accurata

Invece tra gli svantaggi:

- Non può essere usato se si necessita di sistemi completi
- Difficoltà nel identificazione degli incrementi con priorità maggiore

2.5 Modello trasformazionale

I modelli trasformazionali, si basano su specifiche e requisiti scritti in linguaggi di alto livello, spesso di natura **algebrica matematica, insiemistica o con automi a stati finiti**, in questo modo tramite dei processi di trasformazione, modifichiamo questi requisiti astratti man mano in **Codice Eseguibile**.

È basato su due concetti fondamanetali:

- Prototipazione
- Formalizzazione

Poichè con questo modello, si usa un approccio molto formale, diventa difficile produrre software di grandi dimensioni, ma risulta molto comodo, tranne per uno sforzo iniziale , con l'utilizzo di **codice riusabile**.

Per questo si dice che si avvale della **storia dello sviluppo**. Può avvalersi di strumenti software che automatizzano il processo di sviluppo (code generation).

2.6 Modello a spirale

il modello a spirale è un **Meta-modello**, quindi serve per definire altri modelli.

Fulcro di questo modello sono è la **gestione dei rischi**:

> "Identificare, affrontare, ed eliminare i rischi prima che insorgano problemi seri o causa di re-implementazioni costose"

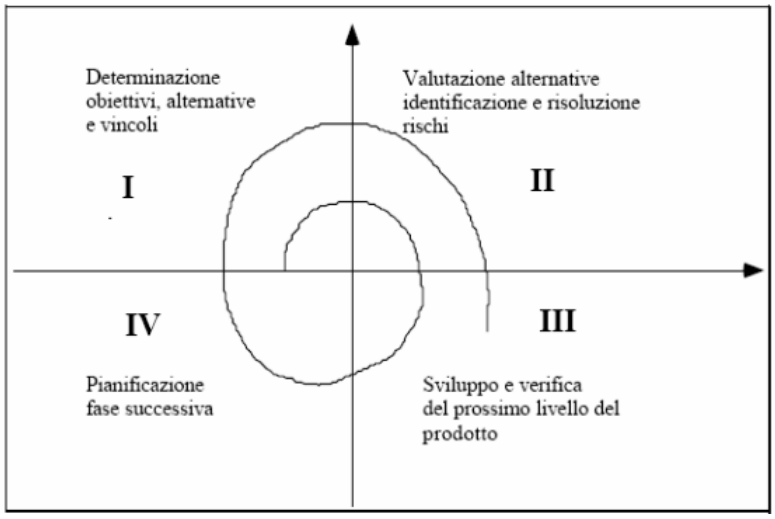

Figura 8: modello a spirale

come possiamo notare è suddiviso in quattro quadranti:

- Determinazione obiettivi, alternative e vincoli:
- Valutazione alternative identificazione e risoluzione dei rischi
- Sviluppo e verifica del prossimo livello del prodotto
- Pianificazione fase succesiva

il **raggio** indica il costo accumulato, quindi è una spirale che cresce verso l'esterno.

ATTENZIONE!!! il modello a spirale è un meta-modello, quindi non esclude l'utilizzo di altri modelli ad esempio un modello evolutivo o a cascata, a seconda dei requisiti.

3 Analisi e specifica dei requisiti

Introduciamo adesso l'analisi e la specifica dei requisiti, è generalmente la prima fase del processo di sviluppo software, ed è anche la fase più importante. In questa fase dobbiamo definire in modo formale, i requisiti della nostra applicazione. Errori in questa fase costano molto caro, infatti si propagano in tutte le altre fasi, con perdita di tempo e denaro.
Le domande principali da farci sono:

- Cosa sono i requisiti?
- Che cosa produce questa fase?

in questo capitolo troveremo le risposte a queste domande.

3.1 Ingegneria dei requisiti

l'ingengeria dei requisiti (requirements engineering), si occupa di **raccogliere , documentare, classificare, analizzare e gestire i requisiti**, cioè le funzionalità, le caratteristiche e i vincoli richiesti al nostro sistema.

L'Ingegneria dei requisiti è di fondamentale importanza, come abbiamo detto errori nella fase della specifica di questi requisiti, si propagano. E senza un approccio **chiaro e sistemico**, è facile comprendere male le richieste con la conseguenza di:

- Clienti non soddisfatti
- Ritardi nella produzione
- Aumento dei costi di manutenzione
- Software che non rispetta le richieste del cliente

3.1.1 I requisiti

I requisiti sono:

> una frase in linguaggio naturale che descrive qualcosa che il sistema dovrà fare **(requisito funzionale)**, o una proprietà o vincolo che si desidera per il sistema **(requisito non funzionale)**, e che uno o più stakeholders (portatori di interesse) richiedono al sistema stesso

Sintetizzando un requisito, esprime a parole, cosa il sistema dovrà fare, le caratteristiche e i vincoli.

3.1.2 Tipi di requisiti

I requisiti si dividono in base alla **astrazione** e al **tipo**.

Astrazione

- I requisiti **utente**, sono requisiti più astratti che descrivono in modo approssimativo le funzionalità del sistema.
- i requisiti **di sistema**, sono meno astratti e vanno a descrivere in modo approfondito, le singole funzionalità.

Tipo

- I requisiti **funzionali**, riguardano i vincoli e le funzionalità richieste al sistema.
- i requisiti **non funzionali**, riguardano gli attributi di qualità che il sistema deve soddisfare, ad esempio il cliente può chiedere un software facile da usare e sarà poi compito del analista di quantificare questo requisito. Ma i requisiti non funzionali riguardano anche i vincoli di sviluppo, tecnologici. (ad esempio un vincolo sul linguaggio di programmazione da usare)
- i requisiti **di dominio**, sono vincoli e richieste che sono proprie del dominio applicativo in cui agirà la nostra applicazione. Ad esempio delle normative vigenti in quell ambito (vedi avionico, ferroviario, etc. . .).

3.2 Analisi, specifica e validazione dei requisiti

Nella fase di **analisi dei requisti** si stabilscono le funzionalità, i servizi e i vincoli richiesti al sistema. Il documento prodotto da questa fase è il DSR (Documento di specifica dei requisiti) ed eventualmente un PTS (piano di testing).

In questa fase si specifica generalmente in linguaggio naturale, è la fase iniziale, in cui i requisiti sono descritti in modo più semplice. Spesso il documento di analisi è svolto da analisti a stretto contatto con gli stakeholders.
Nella fase di **specifica dei requisiti**, c'è un processo di schematizzazione dei requisiti, spesso in un documento strutturato (template)
Nella fase di **validazione** invece, si rivedono le richieste spesso con i Clienti in modo di trovare errori e incongruenze.

3.2.1 Caratteristiche di un SRS(specifica dei requisiti di sistema)

le caratterisctiche di un SRS sono:

1. corretto: Non deve contentere errori

2. completo: Deve contenere tutti i requisiti
3. non ambiguo: I requisiti devono essere esplicitati in modo chiaro senza ambiguità
4. verificabile
5. consistente: I requisiti non devono avere contraddizioni
6. modificabile: una struttura e uno stile facilemnte modificabile
7. tracciabile: sia a posteriori che antecedenti, quindi deve essere facile tracciare il codice a partire dal requisito e viceversa(spesso si usa una numerazione come nello standard IEEE 830-1998)

4 Modellazione

Un modello è un astrazione del nostro sistema, modelliamo il nostro software perchè con il tempo, la dimensione dei sistemi software è cresciuta di molto.

Per questo motivo grazie a queste semplificazioni, riusciamo a modellare sia Il **dominio applicativo** e anche il **dominio della soluzione**

4.1 UML

UML(Uniform Modelling Language) è un linguaggio di modellazione utilizzato in tutte le fasi di sviluppo (analisi, progettazione, etc. . .). Nasce alla fine degli anni 80 dal OMG (Object Management Group). I modelli UML si dividono in :

- Modelli Strutturali o Statici: Modellano gli aspetti statici del sistema, quindi le parti che lo compongono e le interazioni
- Modelli Comportamentali o Dinamici: Descrivono i comportamenti del nostro sistema, ad esempio scenari di utilizzo
- Modelli di interazione: Descrivono le vari interazioni in uno scenario (Sequence Diagram)

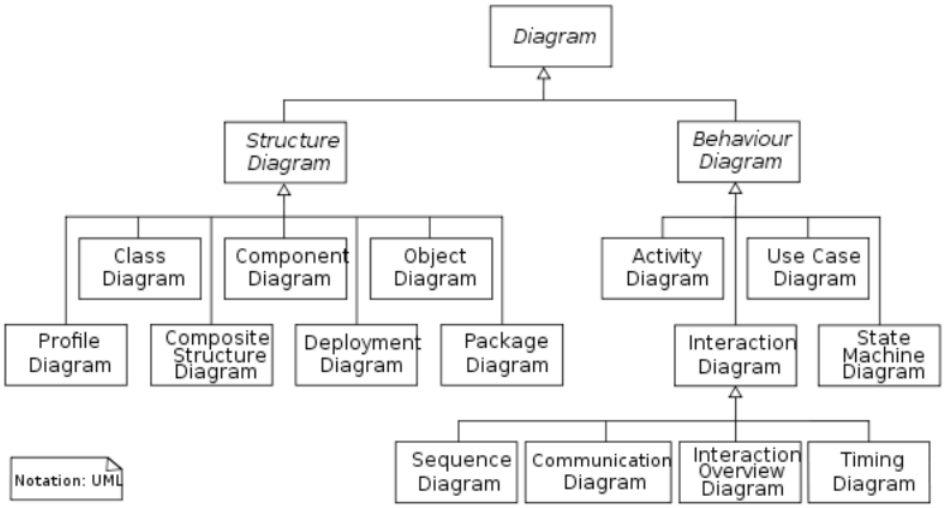

Figura 9: Diagrammi Uml

4.2 Modelli Strutturali

Introduciamo adesso i modelli *strutturali*, che ripetendo, modellano gli aspetti strutturali del nostro sistema ne fanno parte:

- Class Diagram o diagramma delle classi
- Component Diagram
- Object Diagram
- Package Diagram
- Deployment Diagram

4.2.1 Class Diagram

Il diagramma delle classi modella la realta' del nostro sistema.

Il modello segue ampiamente la Object Oriented Programming (OOP).Infatti utilizza ampiamente il concetto di classe, oggetto, eridetarietà e polimorfismo. Il class Diagram viene Utilizzato sia in fase di analisi che di progettazione. infatti distinguiamo in:

- System Domain Model (SDM): Modello concettuale, serve a modellare le classi del nostro sistema, quindi le entita che partecipano alla soluzione
- System Model: Modello più dettagliato, contiene tutte le classi che partecipano alla soluzione, anche quelle di supporto, ad esempio librerie di matematica, librerie per UI, etc...

4.2.2 Object Diagram

L'object diagram e' molto simile al class diagram, almeno nella notazione.

ma invece di modellare la classi e quindi la realta'. Modella una **fotografia** in un istante, del class diagram, quindi modelliamo gli oggetti e le loro interazioni in un particolare scenario.

E' utilizzato per esplicitare in modo chiaro le interazioni tra le classi in uno scenario, quindi e' molto usato nel debugging.

4.2.3 Package Diagram

Il package non e' altro che un contenitore di classi, si usano per racchiudere classi **logicamente**.

Attenzione e' un raggruppamento LOGICO NON FISICO(vedere component diagram).

I package tra di loro condividono lo stesso **Spazio dei nomi**.

4.2.4 Component diagram

Il component diagram e' molto simile al package diagram, ma invece di raccogliere le classi logicamnete, le va a raccogliere fisicamente, quindi modella l'effettiva file structure del progetto .

Particolare importanza e' data al concetto di interfaccia, infatti i COMPONENTI dialogono ctramite interfaccia.

4.2.5 Deployment Diagram

Il deployment diagram mostrano l'architettura fisica del sistema. Gli elementi che modelliamo sono chaiamati NODI(Pc, Server etc...). E sono collegati con archi che rappresentano il canale di comunicazione tra di loro.

4.3 Diagrammi comportamentali

I diagrammi comportamentali, o dinamici, abbiamo detto modellano gli aspetti dinamici, quindi ad esempio scenari di un caso d'uso.

- Use Case Diagram
- Activity Diagram
- State Chart Diagram
- Sequence Diagram
- Communication Diagram
- Interaction Overview Diagram

4.3.1 Use Case Diagram

Un caso d'uso non e' altro che un interazione hw/sw con un attore per raggiungere un risultato.

Un attore non e' altro che una persona, un altro sistema che interagisce col sistema.

Dividiamo in:

- Attore primario: attore che fornisce uno stimolo per avviare il caso d'uso
- Attore secondario: attore che interagisce col sistema dopo l'avvio, quindi una figura secondaria ma comunque necessaaria per la terminazione del caso d'uso

Il caso d'uso pero non esplicita la sequenza di azioni per irsolvere il problema ma solo cosa.

Quindi e' molto generico e vago, e poi con la definizione degli SCENARIO, possiamo definire proprio la sequenza di azioni.

Di parrticolare importanza sono gli scenari alternativi, scenari che avvengono solo in particolari condizioni spesso di errore ad esempio.

Uno degli scopi e' anche quello di definire un CONFINE PER IL MIO SISTEMA, cioe' quali sono le funzionalita' da produrre, responsabilita' del sistema, e cosa e' esterno al sistema.

4.3.2 Sequence diagram

Il sequence diagram e' un modello dinamico, che esplicita le sequenze di azioni da svolgere per terminare un caso d'uso. Quindi partiamo da uno o piu' attori e poi le vari interazioni tra gli oggetti del sistema.

Il sequence puo' essere usato sia in fase di analisi che di progettazione(sequence di dettaglio). Importante nel sequence e' l'esplicitazione del tempo!!.

4.3.3 Communication diagram

Molto simile al sequence, infatti utilizza le stesse notazioni, ma invece di esplicitare le azioni in verticale. Il tempo viene rappresentato da una NUMERAZIONE dei messaggi.

4.3.4 State Chart Diagram