

Ingegneria dei software

dmotrio.xyz

7 aprile 2022

Indice

1	T1 - Software Development Process	3
1.1	Cos'è l'ingegneria del software?	3
1.1.1	Processo del software	3
1.2	Modelli di sviluppo del software	3
1.2.1	Modello a cascata	3
1.2.2	Modello a spirale	4
1.2.3	Sviluppo incrementale	4
1.2.4	Sviluppo guidato dai test	5
1.2.5	Sviluppo agile	5
1.2.6	Extrem programming	5
1.3	Software riusabile	5
2	T2 - Coding, Debugging, Testing	5
2.1	Legge di Ambler per gli standard	5
2.2	Coding practices	6
2.3	Gestione degli errori	6
2.4	Testing	6
2.4.1	Unit testing	6
2.4.2	Integration testing	6
2.4.3	System testing	6
2.4.4	Functional testing	6
2.4.5	Performance testing	6
2.4.6	Acceptance testing	6
3	T3 - System Modelling and UML	7
3.1	UML	7
4	T4 - Requirements engineering	7
4.1	Classificazione dei requirements	7
4.1.1	Requisiti funzionali(features di sistema)	7
4.1.2	Requisiti non funzionali(features di sistema)	7
4.1.3	Requisiti del dominio(features di sistema)	7
4.1.4	Requisiti volatili(natura statica/dinamica)	7
4.2	Rischi	7
4.3	Documento di specifica dei requirements	8
5	T5 - Requirements engineering and UML	8
5.1	Use case diagram	8
5.2	Class diagram	8
5.2.1	Associazione	8
5.2.2	Generalizzazione e nesting	9
5.2.3	Dipendenza e realizzazione	9
5.2.4	Aggregazione e composizione	9
5.3	Sequence diagram	9
5.4	Activity diagram	9
5.5	Robustness diagram	9
6	T6 - Feasability and requirements elicitation	9

Elenco delle figure

1	Modello a cascata	3
2	Modello a spirale	4
3	Modello a incrementale	5
4	Modello a extrem programming	5
5	Associazioni	8
6	generalizzazione e nesting	9
7	Aggregazione e composizione	9

Elenco delle tabelle

1 T1 - Software Development Process

L'Ingegneria del Software non è solo scrivere codice è piuttosto un concetto di risoluzione dei problemi del mondo reale sfruttando il software.

I requisiti sono sempre più stringenti: tempi brevi, sistemi complessi, molte features.

Un buon software deve avere ottime **maintenability, dependability, efficiency, acceptability**.

I problemi e le soluzioni sono sempre complesse ma il software permette massima flessibilità. È un sistema discreto.

Le sfide principali sono rappresentate da **eterogeneità, delivery, trust**.

La fase di problem solving si suddivide in analisi e sintesi.

1.1 Cos'è l'ingegneria del software?

L'Ingegneria del software è un **insieme di tecniche, metodologie, strumenti** che aiutano nella produzione di software di alta qualità dati un budget, una scadenza, e delle modifiche continue.

La sfida principale è quella di aver a che fare con complessità elevate e ad un aumento delle responsabilità, dato che un ingegnere del software non deve solo scrivere codice, ma piuttosto lavorare con competenza e confidenzialità, attenendosi ad un'etica.

1.1.1 Processo del software

Dopo una rappresentazione astratta, si procede con un set di attività strutturate: specifiche dei **requirements, design, implementazione, validazione, evoluzione**.

1.2 Modelli di sviluppo del software

Distinguiamo tra **plan-driven** e **agile** development. Nel primo si pianificano i requisiti e solo in seguito si sviluppa il software. Nel secondo si sviluppa il software un pezzo alla volta, a stretto contatto con il cliente per dei feedback.

1.2.1 Modello a cascata

Modello plan-driven, le specifiche e lo sviluppo sono separati.

I pro:

- ottima documentazione
- manutenzione semplice

I cons:

- specifiche congelate dopo la pianificazione iniziale
- cliente poco coinvolto
- tempi lunghi

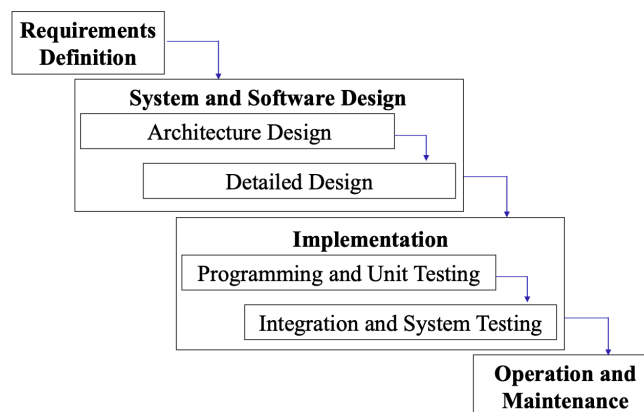


Figura 1: Modello a cascata

1.2.2 Modello a spirale

Diverse fasi che susseguono a spirale, la gestione del rischio viene gestita tramite prototipazione che permette di testare i prodotti.

I pro:

- prevenzione dei rischi
- completezza della documentazione
- flessibilità
- elevata usabilità
- buon design
- facilità di manutenzione
- ridotto costo di sviluppo

I cons:

- modello costoso
- riservato ad esperti e a progetti costosi

Il prototipo è un'implementazione limitata del sistema, rappresentanti solo alcuni aspetti alla volta.

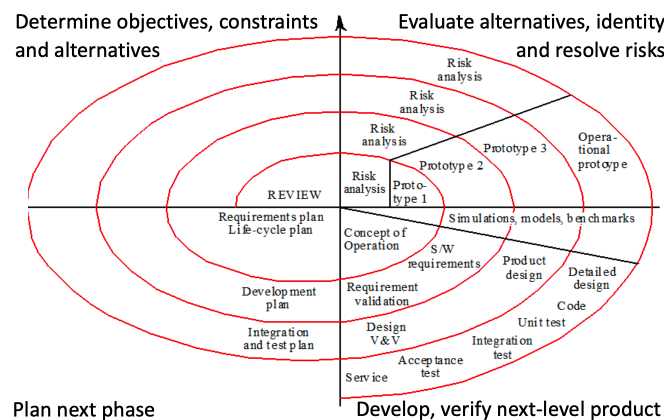


Figura 2: Modello a spirale

1.2.3 Sviluppo incrementale

Modello che si suddivide in fasi:

1. raccolta dei requisiti
2. versione iniziale
3. fase di design
4. fase di implementazione
5. produzione di versione finale

I pro:

- naturale presenza di prototipi ad ogni aggiunta di features
- basso rischio di fallimento
- qualità di testing in base alla priorità

I cons:

- scarsa visibilità d'insieme
- sistemi mal strutturati
- skill speciali necessarie

Adatto a progetti piccoli o parti di progetti grandi.

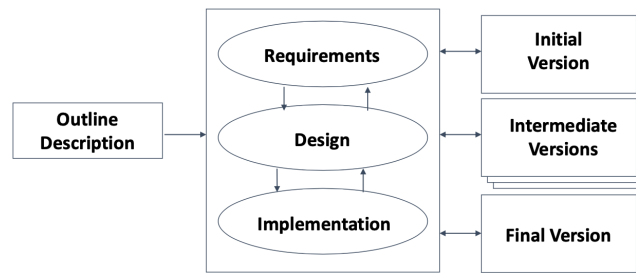


Figura 3: Modello a incrementale

1.2.4 Sviluppo guidato dai test

Vengono prima scritti i test e poi l'implementazione mettendo le difficoltà in primo piano.

Rende il debug più semplice.

Si aggiunge il nuovo test e poi la feature così da verificare che tutti i test precedenti siano validi.

1.2.5 Sviluppo agile

Questo modello di sviluppo è basato sul concetto di delivery del prodotto continuo avendo delle feature in continua evoluzione.

Il cliente è al centro dello sviluppo, il team si autoorganizza e le feature vengono aggiunte man mano.

Essendoci la mancanza di planning il team deve essere esperto per non perdersi.

Spesso si creano documentazioni sbagliate o incomplete.

1.2.6 Extrem programming

L'XP viene scelta quando i requisiti cambiano velocemente, i team sono ridotti e affiatati (pair programming per esempio).

Tipo di programmazione agile, basato sul design semplice, release minori, refactoring continuo, alta semplicità.

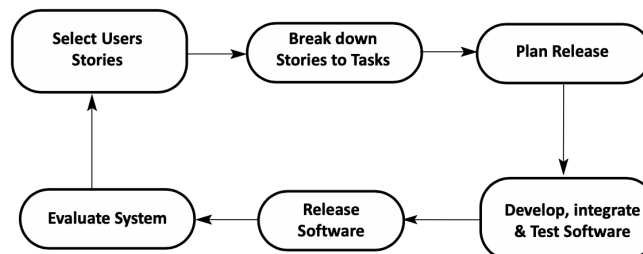


Figura 4: Modello a extrem programming

1.3 Software riusabile

È comodo lavorare per **microservizi atomici**, di modo da usarli in seguito se si affronta un problema simile, un'esempio sono le API, è sempre meglio riusare qualcosa che riscriverlo da zero.

Questo "mindset" riduce i tempi e i costi di sviluppo nel lungo periodo.

Il contro è il fatto che i microservizi non sono fatti espressamente per una task specifica e si va a sacrificare qualche requisito.

2 T2 - Coding, Debugging, Testing

Lo stile di coding è importante perché un programma viene scritto una volta e letto spesso da altri!

Stare attenti al layout, nomi, commenti, ecc...

2.1 Legge di Ambler per gli standard

Più uno standard è adottato e più sarà facile farlo usare al proprio team.

Non perdere tempo adattare a standard scemi.

Se hai dei dubbi usa gli standard di google che sicuramente ne sanno più di me e te. [standard di google](#)

Solitamente si hanno degli standard aziendali e questi standard vanno a chiarire aspetti come:

- nomenclatura
- formattazione
- formato
- contenuto

2.2 Coding practices

- indentazione
- whitespaces
- naming, commenting

Fondamentale è spiegare i compiti delle classi, funzioni e processi complessi.

Utilizzare uno standard permette ai colleghi di non dover riscrivere il codice. Rendendo più semplice le iterazioni del codice successive.

2.3 Gestione degli errori

Si fa un distinguo fra il prima, durante e dopo:

- prevenzione
- rilevamento
- recupero

Per debuggare, bisogna riconoscere l'errore, isolare la fonte, identificarne la causa, trovare un fix, applicarlo e testarlo.

2.4 Testing

Permette di trovare errori ma non la loro assenza, **il tester non dovrebbe essere il programmatore.**

2.4.1 Unit testing

Test di singoli componenti.

2.4.2 Integration testing

L'intero sistema è visto come insieme di sottosistemi, l'obiettivo è quello di testare tutte le interfacce e le interazioni fra i sottosistemi.

2.4.3 System testing

test dell'intero sistema per vedere se rispetta i requisiti funzionali.

2.4.4 Functional testing

Verifica delle funzionalità del sistema, test basati sui requisiti del progetto.

2.4.5 Performance testing

Test in situazioni estreme(stress testing, volume testing, recovery testing, penetration testing).

2.4.6 Acceptance testing

Il sistema è pronto per la produzione??

Test scelti ed effettuati dal cliente.(alpha e beta test)

3 T3 - System Modelling and UML

Rappresentazione astratta del sistema e dei problemi, si devono introdurre i componenti essenziali mediante una nozione consistente.

Il system modelling deve essere **predictive**(prima del development), **extracted**(da un sistema esistente) e **prescriptive**(definire regole per l'evoluzione del software).

3.1 UML

UML è semplice, espressivo, utile, consistent, estensibile.

4 T4 - Requirements engineering

Gli scopi dell'ingegnerizzazione dei requisiti è **identificare** i servizi necessari e i constraint, **definire** offerta e contratto, **ottenere** tutte le informazioni necessarie al design.

Si cerca di ottenere requirements:

- validi(reali necessità)
- non ambigui
- completi
- comprensibili
- consistenti
- prioritizzati
- verificabili
- modificabili
- tracciabili

4.1 Classificazione dei requirements

4.1.1 Requisiti funzionali(features di sistema)

Descrivono funzionalità di sistema o di servizio, come input di dati, output, operazioni svolte, workflow, autorizzazioni.

4.1.2 Requisiti non funzionali(features di sistema)

Descrivono i limiti di parti del sistema e del suo sviluppo. Specificano criteri per giudicare l'operato del sistema.

4.1.3 Requisiti del dominio(features di sistema)

Derivato dal campo di utilizzo del software.

4.1.4 Requisiti volatili(natura statica/dinamica)

- mutable requirements = cambiano nel tempo(tasse, normative, ecc)
- emergent requirements = cambiano quando il cliente capisce di più il sistema
- consequential requirements = emergono con l'informatizzazione del sistema che non lo era
- compatibility requirements = emergono dal dover interfacciare il sistema con altri sistemi

4.2 Rischi

I rischi della stesura dei requisiti possono essere:

- imprecisioni
- conflitti tra requisiti

4.3 Documento di specifica dei requirements

Questo documento specifica i requisiti del sistema, includendo una definizione e una specifica.

Prende il nome di system specification se include direttive su hardware e software.

Software Requirements Specification (**SRS**) se include solo specifiche software.

Un **SRS** deve includere:

- introduzione
- descrizione generale
- features
- requirements

Il linguaggio naturale usato per stilare questo documento implica ambiguità, per questo si ricorre a una struttura ben definita per evitare ambiguità.

5 T5 - Requirements engineering and UML

5.1 Use case diagram

In questo diagramm vengono inclusi tutti i casi di utilizzo del sistema, da parte dei vari attori che rappresentano gli utenti.

Il **System boundary** divide l'interno dall'esterno del sistema.

5.2 Class diagram

Diagramma che rappresenta le classi tramite:

- gli attributi (pubblici con +, privati con - e # per i protetti)
- le funzioni delle classi

5.2.1 Associazione

Quando una classe svolge il ruolo di variabile all'interno di un'altra classe, questa connessione deve essere rappresentata, si usano:

- cardinalità
- direzione
- constraint
- ruoli

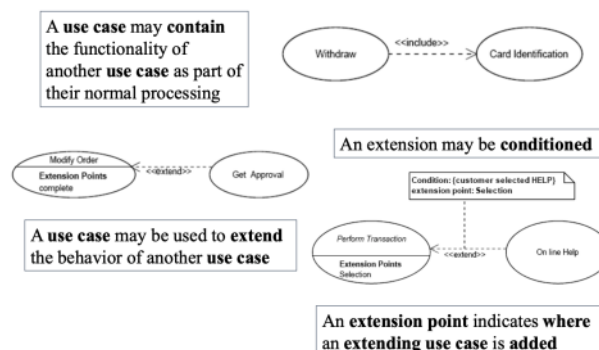


Figura 5: Associazioni

5.2.2 Generalizzazione e nesting

La generalizzazione indica **ereditarietà**, nesting indica che una classe è nestata nella classe dove arriva l'operatore.

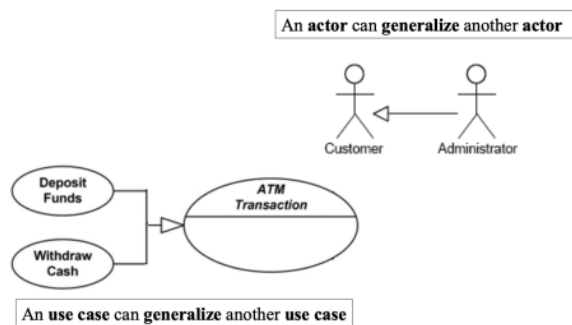


Figura 6: generalizzazione e nesting

5.2.3 Dipendenza e realizzazione

- **dipendenza** = relazione debole tra client e supplier
- **realizzazione** = relazione tra specifica e implementazione

5.2.4 Aggregazione e composizione

L'aggregazione rappresenta un elemento composto da altri elementi minori.

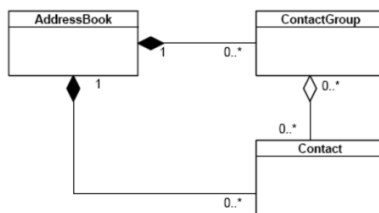


Figura 7: Aggregazione e composizione

5.3 Sequence diagram

Usa un timeline verticale per rappresentare l'interazione tra le classi. Ci possono essere comunicazioni sincrone e asincrone.

5.4 Activity diagram

Diagramma che rappresenta un'operazione eseguita nel sistema con rappresentazione delle risorse utilizzate (io, disk, banda, ecc)

5.5 Robustness diagram

Diagramma UML semplificato che appresenta gli use cases verificandone la correttezza, completezza e requisiti.

6 T6 - Feasability and requirements elicitation