Ingegneria dei software

kanopo

11 aprile 2022

Indice

Elenco delle figure

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 sitesi.

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 quelli 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 coninvolto
- tempi lunghi

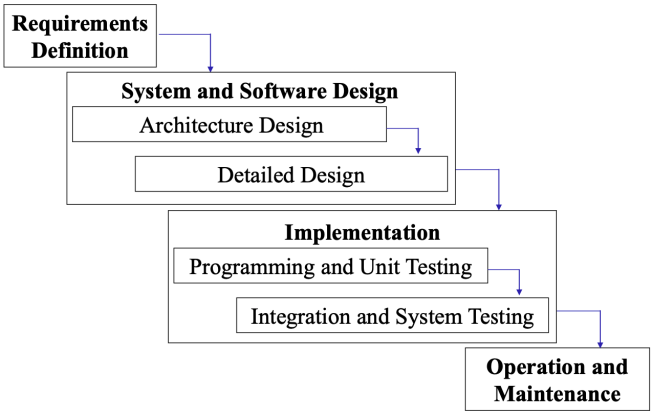

Figura 1: Modello a cascata

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.

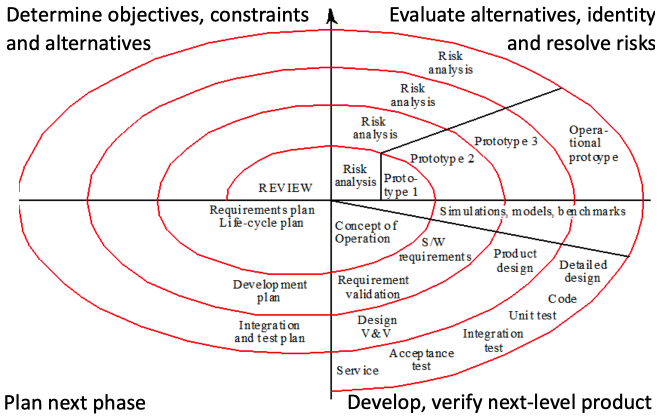

Figura 2: Modello a spirale

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.

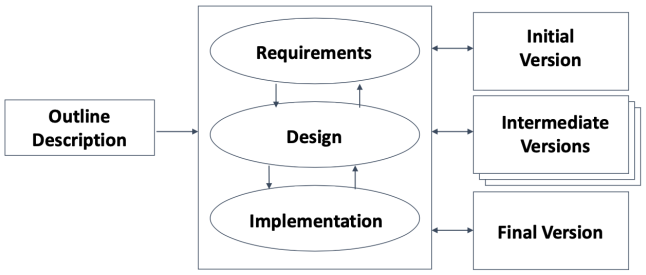

Figura 3: Modello a incrementale

1.2.4 Sviluppo guidato dai test

Vengono prima scritti i test e poi l'implementazione mettendo le difficolta in primo piano.
Rende il debug più semplice.
Si aggiunte il nuovo test e poi la features cosi da veriicare 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 features in continua evoluzione.
Il cliente è al centro dello sviluppo, il team si autoorganizza e le features 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 scelte quado 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.

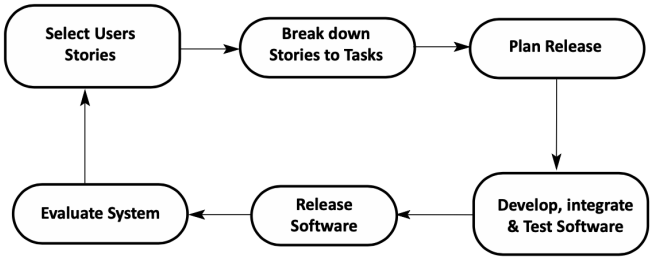

Figura 4: Modello a extrem programming

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 temi e i costi di sviluppo nel luno 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 perchè un programma viene scritto una volta e letto spesso da altri!
Stare a ttenti ai 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 adietro a standard scemi.
Se hai dei dubbi usa gli standaard 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

Fondaentale è spiegare i compiti delle classi, funzioni e procecssi complessi.
Utilizzare uno standard permette ai collechi 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:

- prrevenzione
- 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 sottositemi, l'obbiettivo è quello di testare tutte le interfacce e le interazioni fra i sottositemi.

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 dedvono introdurre i componenti essenziali mediante una noazione 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 constrait, **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 limitti 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, nomative, 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 harware 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 ecgineering 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 ruole di variabile all'interno di un'altra classe, questa connessione deve essere rappresentata, si usano:

- cardinalità
- direzione
- constraint
- ruoli

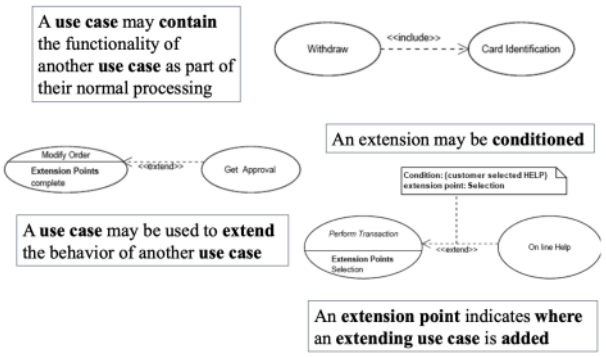

Figura 5: Associazioni

5.2.2 Gneralizzazione e nesting

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

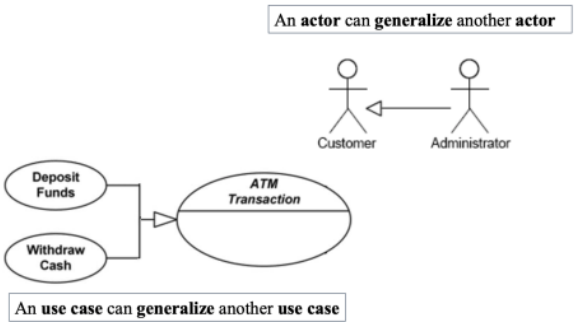

Figura 6: generalizzazione e nesting

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.

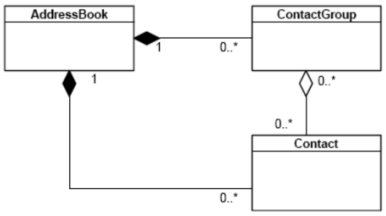

Figura 7: Aggregazione e composizione

5.3 Sequence diagram

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

5.4 Activity diagram

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

5.5 Robustnes diagram

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

6 T6 - Feasability and requirements elicitation

Per ottenere le informazioni necessarie dobbiamo identificare le fonti, acquisire i dati, verificarli e sintetizzarli.
Le varie persone interessate al progetto possono essere suddivise cosi:

- stakeholder: interessato nel progetto
- developer: produce con pochi errori
- project managment: budget, scadenze

- investor: velocizzazione del progetto
- cliente e utente: usabilità e workflow

6.1 Tecniche di elicitation

(elicitation = tirare fuori le informazioni)

6.1.1 Document analysis

Analizzare i documenti e insieme agli stakeholder verificare la correttezza dei dati.

6.1.2 Observation of the work environment

Apprendere informazioni importanti osservando il lavoratore.

6.1.3 Questionario

Determinare i fatti e leopinioni con un questionario.

6.1.4 Interviste

Decidere chi intervistare e acquisire i dati.

6.1.5 Scenari e casi di utilizzo

Esempi IRL(In Real Life) di come il sistema verrà usato.

6.2 Attività di supporto all'elicitation

6.2.1 Brainstorming

composto dalle fasi storm(spara idee) e calm(analizza idee)

6.2.2 Focus group

simile al brainstorming

6.2.3 Prototipi

Esecuzone di una task e si analizza lasituazione facendo emergere problemi e soluzioni.

7 T7 - Use cases

Gli use cases sono scenari plausibili che sfruttano i diagrammi UML.
Ogni requirements deve essere mappato da almeno un use case.

7.1 Componenti principali

Lo use case deve definire lo stato precedente del sistema, l'ordine degli eventi, le alternative, situazioni eccezionali e i risulati. Deve menzionare gli attori che prendono parte al sistema, i problemi di design, i diagrammi di relazione.
Ecco delel guidelines:

- non pensare all'implementazione
- essere pessimistici
- elencae gli scenari funzionanti
- elencare tutti i possibili use cases
- utilizzare un formaato standard
- utilizzare verbi appropriati
- documentare bene le situazioni eccezionali
- non rappresentare singoli step come use cases

Name	The Use Case name. Typically the name is of the format <action> + <object>.
ID	An identifier that is unique to each Use Case.
Description	A brief sentence that states what the user wants to be able to do and what benefit he will derive.
Actors	The type of user who interacts with the system to accomplish the task. Actors are identified by role name.
Organizational Benefits	The value the organization expects to receive from having the functionality described. Ideally this is a link directly to a Business Objective.
Frequency of Use	How often the Use Case is executed.
Triggers	Concrete actions made by the user within the system to start the Use Case.
Preconditions	Any states that the system must be in or conditions that must be met before the Use Case is started.
Postconditions	Any states that the system must be in or conditions that must be met after the Use Case is completed successfully. These will be met if the Main Course or any Alternate Courses are followed. Some Exceptions may result in failure to meet the Postconditions.
Main Course	The most common path of interactions between the user and the system. 1. Step 1 2. Step 2
Alternate Courses	Alternate paths through the system. AC1: <condition for the alternate to be called> 1. Step 1 2. Step 2 AC2: <condition for the alternate to be called> 1. Step 1
Exceptions	Exception handling by the system. EX1: <condition for the exception to be called> 1. Step 1 2. Step 2 EX2 <condition for the exception to be called> 1. Step 1

Figura 8: Tamplate per i casi di utilizzo

8 T8 - Requirements

L'analisi dei requisiti permette di raffinare e strutturare i requisiti in modo da renderli più chiari, precisi e formali.

8.1 Analysis classes

Il concetto è quello di astrarre le entità del problema.

8.2 Classes discovering techniques

8.2.1 Noun verb analysis

Analisi che sfrutta i contenuti delle specifiche del progetto. Necessita di completezza del documento molto alta.

8.2.2 Use case driven approch

Approccio che sfrutta fli scenari degli use cases.

8.2.3 Common class patterns

Analisi basata sulla teoria della classificazione generica degli oggetti.
Fornisce linee guida, ma non un processo sistematico per ottenere le classi.
Introduce errori di mal'interpretazioni.

8.2.4 CRC cards

Sono usate in specifiche sessioni di brainstorming, generalmente si parte dagli use cases e s icreano delle card con:

- class name
- responsabilities
- collaborators

Non è un metodo sistematico, si usano le CRC cards come validazione.
Gli achievement di questo metodo sono:

- Verifica la correttezza dello use case
- verifica la correttezza delle associazioni
- verifica la correttezza delle generalizzazioni

- trovare le classi omesse
- scovare opportunità di refactoring

8.2.5 Mixed approch

Migliore:

1. Le classi iniziali provengono dalla conoscenza del dominio
2. si sfrutta come guida il common class pattern
3. per aggiungere altre classi si usa la noun verb analysis
4. per verificare il lavoro si usano gli use cases
5. per il brainstorming si usano le CRC cards

9 T9 - Requirement validation and managment

9.1 Valdation

La validazione consiste nella verifica della correttezza dei requisiti, con obbiettivi la completezza e i costi.

- consistenza
- realismo
- verificabilità

9.2 Requirements managment

Si verificano gli errori, conflitti e inconsitenze.
Soddisfare il cliente e mantenere i costi bassi:

- identificazione dei requisiti
- processo di modifica
- policy di tracciabilità
- supporto del case tool

9.2.1 Case tool support

non ho voglia di spiegarlo.

9.2.2 Stati dei requirement

tutto questo ambaradam è molto simil ad un issue tracker come git.

- proposal
- approved
- rejected
- implementation
- verified
- deleted

Si fanno sostanzialemnte dei test con un version control e si fa il roll back in caso di problemi.

9.2.3 Tracciabilità

Anche questo discorso viene gestito bene da un software tipo git.

9.2.4 Traciability planning

Elemeti fondamentali del planning:

- tipi di stakeholder
- informazioni richieste
- dove e da chi raccogniere info
- dove e come mantenere le info
- dove e come cercare le info

Ci sono delle limitazioni per il plannign:

- numero di requirements
- lifetime stimato
- livello di maturità dell'azienda
- dimensione progetto
- altri vincoli dell'utente

12 T12 - Object constraint lenguage

Fa parte delle specifiche UML, e segue uno standard di IBM, è un linguaggio formale usato per descrivere i modelli Object oriented.

12.1 Model type

sono classi, subclasses, classi di associazione, interfacce ed enumerazioni.
Hanno proprietà e attributi.

12.2 Operazioni, espressioni, constraint

I constraint sono dei valori booleani che veificano una condizioni, servono a porre limitazioni sul tipo di valore che può avere una classe.

- Migliorano la documentazione
- più precisione
- comunicazione senza ambiguità

Vari tipi di constraint:

- invarianti(quando l'istanza è a riposo)
- precondizionali
- postcondizionali
- guard(vera prima e dopo)

13 T13 - Design process

Processo che permette di capire come implementare il sistema.
Si parte dai requirements e si tiene conto dei design general princilas.

13.1 Linee guida

- esibire organizzazione modulare che fa uso intelligente di controllo tra componenti
- partizione in componenti logici
- descrivere sia dati che procedure
- arrivare ad interfacce che riducono la complessità

Un buon design deve implementare tutti i requirement espliciti del modello di analisi, e tutti quelli impliciti desiderati dal cliente. Dev'essere leggibile e comprensibile, sia per chi scrive codice, sia per chi lo testa e supporta. In generale, dovrebbe dare un'immagine generale del software, indicandone i dati, i domini funzionali e behavioral dal punto di vista dell'implementazione.

13.2 Stadi del design process

1. comprendere il problema
2. identificare una o più soluzioni
3. processo iterativo ed incrementale

13.3 Design as series of decisions

Alcuni tradeoff possono essere:

- funzionalità vs usabilità
- costo vs robustezza
- efficienza vs portabilità
- velocità di sviluppo vs funzionalità
- costo vs riutilizzabilità

Gli step per una decisione pesata sulle priorità sono:

1. Elencare le alternative possibili
2. elencare i pro e i contro di ogni alternativa(rispetto agli obiettivi e alle priorità)
3. determinare se alcune alternative impediscono altri obiettivi
4. scegliere l'alternativa che permette di raggiungere più obiettivi
5. aggiustare la priorità per decisioni future

13.3.1 Approcci

Approccio **top-down**: prima si fa il design ad alto livello del sistema, poi si prendono le decisioni man mano che si sviluppa il sistema.
Approccio **botton-up**: prima si decidono le utilities di basso livello riutilizzabili, poi il modo di integrarle insieme.
Approccio **mixed**: si usa top-down per scegliere la struttura ed il bottom-up per i componenti riutilizzabili.

13.3.2 Attività, rischi e obiettivi

La fase di design ha tre fasi:

- design dell'architettura
- user interface
- class design

Un buon design è **flessibile, dettagliato, ben documentato e buona possibilità di managment**.

14 T14 - Design concepts

Un sistema sftware ha diversi possibili problemi:

- rigidità
- fragilità
- immobilità(difficle riusare i componenti)
- viscosità(più semplice l'hack del mantenimento del design originale)

14.1 Software design concepts

14.1.1 Abstraction

L'astrazione permette di focalizzarsi su parti importanti del software senza perdersi nell'implementazione. Permette la descrizione di un sistema come struttura a livelli.

14.1.2 Refinement

processo top-down iterabile che ad ogni passo trasforma istruzioni generiche in istruzioni più specifiche.

14.1.3 Nascondere le informazioni

L'information hiding è strettamente legato a:

- astrazione
- coupling
- coesione

14.1.4 Modularità

Un metodo di design può essere detto modulare solo se supporta:

- decomposability(ridurre in modulli più piccoli)
- composability(accorpare in un unico modulo)
- understandability(un modulo è comprensibile senza sapere gli altri)
- continuity(la modifica in un modulo modifica solo lui)
- protection(bug in un modulo?? affligge solo qul modulo!)

I principi per il design modulare sono: linguistic modular units (i moduli devono corrispondere alle unità del linguaggio, come pacchetti o moduli), poche interfacce, poco scambio di informazioni tra moduli, interfacce esplicite (se due moduli comunicano, dev'essere ovvio), information hiding(tutte le informazioni di un modulo dovrebbero essere private, se non specificatamente dichiarato il contrario, e per l'accesso bisogna utilizzare interfacce).

14.1.5 Coesione

La coesione misura la chiusura di una relazione tra elementi di un componente/classe. Una coesione forte è desiderabile perché semplifica le correzioni, le modifiche, le estensioni, riduce il testing e promuove il riutilizzo. Abbiamo più livelli di coesione (dal più basso al più alto):

- coincidental(elementi senza relazioni ma uniti per convenienza)
- logical(elementi che fanno cose simili)
- temporal(elementi attivi nello stesso tempo)
- procedural(elementi che compongono una singola sequnza di controllo)
- communicational(elementi con stesso input e stesso output)
- sequential(l'output di uno è l'input dell'altro)
- functional(elementi che soppriscono ad un funtiona lrequirement)
- object(operazioni tra oggetti)

14.1.6 Coupling

Misura l'interconnessione tra moduli. Quando è debole, i moduli sono fortemente indipendenti. Livelli di coupling dal più debole al più forte:

- no direct(nessuna dipendenza)
- data(solo dati necessari)
- stamp(passati una lista di argomenti ma solo alcuni vengono usati)
- control(si passano flags e altri paramentri)
- externl(legato a device o device drivers)
- common(uso di dati globali)
- content(modificadi dati dell'altro modulo)

Con coupling elevato i componenti sono difficili da comprendere autonomamente, le modifiche causano bug in altri moduli e il riuso di moduli fa schifo.
Con coupling basso aumentano i costi di performance ma lo sviluppo ne gode.

15 T15 - Objects oriente design principles

15.1 Principi dell'Object Oriented Design

15.1.1 SRP - Single Responsability Principle

Una classe deve avere una sola ragione di cambiare.
Le modifiche ai requirements si mappano sulle responsabilità.
Avere tante classi con responsabilità separate semplifica e rende il design flessibile.

> *In programming, the Single Responsibility Principle states that every module or class should have responsibility over a single part of the functionality provided by the software.*

15.1.2 OCP - Open Closed Principle

Un modulo deve essere aperto verso l'esterno ma chiuso all modifica.
Meglio implementare cose in classi derivate piuttosto che modificae una classe.

> *In programming, the open/closed principle states that software entities (classes, modules, functions, etc.) should be open for extensions, but closed for modification. If you have a general understanding of OOP, you probably already know about polymorphism. We can make sure that our code is compliant with the open/closed principle by utilizing inheritance and/or implementing interfaces that enable classes to polymorphically substitute for each other.*

15.1.3 Liskov Substituotion Principle

Le classi derivate devono adempiere alle funzioni ereditate dalle classi padre altrimenti se si usa una classe derivata al posto delal padre si generano errori.

> *More generally it states that objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.*

15.1.4 ISP - Interface Segregation Principles

Avere tante interfacce client-specific è meglio di una generica.

> *In programming, the interface segregation principle states that no client should be forced to depend on methods it does not use. Put more simply: Do not add additional functionality to an existing interface by adding new methods. Instead, create a new interface and let your class implement multiple interfaces if needed.*

15.1.5 DIP - Dependency Inversion Principles

I moduli di alto livello non dovrebbero dipendere da quelli di basso livello ma dovrebbero dipendere da astrazioni.

15.2 Principi di package cohesion

15.2.1 REP - Release/reuse Equivalency Principle

Per riusare un componente non si fa copia incolla(perchè si perdono le modifiche alla libbreria) ma si inserisce il codice mediante una release della libreria e poi si modifica un'istanza.

15.2.2 CCP - Common Closure Princple

Le classi che cambiano insieme devono rimanere insieme.

15.2.3 CRP - Common Reuse Principle

Le classi che non sono reutilizzabili insieme non dovrebbero essere pacchettizate insieme.

15.3 Package Coupling Principle

15.3.1 ADP - Acyclic Dependencies Principle

No cicli nel grafico delle dipendenze.

15.3.2 SDP - Stabel Dependencies Principle

Ogni volta che un pacchetto cambia, tutti quelli che dipendono da lui devono essere validati.

Ovviamente tu preferisci i pacchetti stabili

15.3.3 SAP - Stable Abstraction Principle

Il pacchetto perfetto è tanto stabile quanto astratto.
I pacchetti astratti dovrebbero essere responsabili e indipendenti, mentre quelli concreti dovrebbero essere irresponsabili e dipendenti.

15.4 Attività per un buon design

- Dividi e conquista(scompore tutto in processi semplici da fare uno alla volta)
- Aumenta la coesione(Un sistema con alta coesione tiene insieme quello che deve stare insieme e lascio fuori il resto)
- Ridurre il coupling(meno interdipendenze tra moduli)
- Aumenta l'astrazione(meglio nascondere i dettagli e ridurre la complessità)
- Aumentare la riusabilità
- Riutilizzare design e codice esistente
- Programmare con l'idea di flessibilità(modifiche future)
- Anticipare l'obsolescenza(pensare alle tecnologie e pianificare la gestione del sistema)
- pensare alla portabilità
- pensare alla testabilità
- sviluppare con l'idea di difendere il software dal'utente.

16 T16 - Design Patterns

Un design pattern è una soluzione ricorrente ad un problema comune, che può essere usata in modi diversi. È dipendente da linguaggio ed implementazione. I design patterns hanno vari obiettivi:

- Codificare un buon design
- dare nomi espliciti alle strutture
- ciao